우리식조작체계문고

JAVA 프로그람작성법

교육성교육정보쎈터
주체97(2008)

차 례

머 리 말

위대한 수령 **김일성**동지께서는 다음과 같이 교시하시였다.

《새로운 과학분야를 개척하며 최신과학기술의 성과를 인민경제에 널리 받아들이기 위한 연구사업을 전망성있게 하여야 합니다.》(《**김일성**저작집》 제35권, 328페지)

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《프로그람을 개발하는데서 기본은 우리 식의 프로그람을 개발하는것입니다.》(《**김정일**선집》 제15권, 196페지)

위대한 령도자 **김정일**동지의 현명한 령도에 의하여 오늘 우리 나라에서는 정보산업이 비약적으로 발전하고있다. 정보기술, 프로그람기술이 과학연구뿐아니라 생산과 경영활동을 비롯한 사회생활의 모든 분야에 광범히 도입되여 응용되고있으며 경제적효과성을 높이고있다.

콤퓨터망이 전국적범위에서 형성되고있고 나라의 정보화가 적극 추진되고있는 오늘의 현실은 정보화실현을 위한 망관련프로그람들을 많이 개발할것을 요구하고있다.

Java언어는 90년대에 출현하여 많은 프로그람개발자들과 과학자, 기술자들이 관심을 가지고 활발히 활용하고있는 망기반의 프로그람작성언어이다. Java언어는 객체지향언어로서 원천프로그람만 있으면 아무런 기반에서나 실행시킬수 있다는 특징을 가지고있다. 최근 쏘프트웨어공학과 기술이 빨리 발전하면서 현실에서 제기되는 기술공학적모형들을 콤퓨터론리모형으로 전환하고 규모가 큰 개발대상들을 해결하는데서 Java언어의 역할이 보다 높아지고있다. 현재 Java언어는 객체에 대한 요구분석, 설계, 실현의 쏘프트웨어개발분야에서 아주 적합한 전망성있는 언어이다.

특히 우리 식 조작체계에 의한 정보체계가 구축되고있는 환경에서 Java언어는 기존 프로그람들을 쉽게 새로운 조작체계에 이식시킬수 있는 전환성이 높은 언어로 된다.

이 책은 Java프로그람작성법의 기초편이다. 책에서는 Java언어에 대한 개념으로부터 출발하여 프로그람작성기초, 클라스의 개념 그리고 여러가지 측면에서의 프로그람작성수법들을 실례를 들어 서술하였다.

제1장. Java프로그람작성초보

이 장에서는 객체와 클라스의 개념, 객체의 속성과 객체들사이의 호상관계, Java 프로그람개발의 기본단계와 Java프로그람의 구성, 기본입출력프로그람작성 및 Java 언어의 주요특징에 대하여 서술한다.

제1절. 객체와 클라스

- 객체지향기술은 인간의 사유방식을 쏘프트웨어에 도입한것이다.
- 객체는 현실세계의 실체나 개념을 추상화한것이다.
- 객체지향의 3대요소는 클라스, 객체, 통보이다.
- 객체지향의 3대원리는 추상, 내장, 계승이다.
- 클라스는 같은 종류의 객체들의 모임이며 클라스의 실체화의 결과가 객체이다.
- 실체는 객체의 구체적인 표현이다.

1.1.1. 객체지향방법론이 나오기까지

초기 콤퓨터에서 실행하는 프로그람은 대부분 특정한 하드웨어체계의 전문설계를 위한것이였다. 이것을 **기계지향프로그람**이라고 부른다. 이러한 프로그람들의 실행속도와 효률은 아주 높다. 그러나 이러한 기계지향프로그람은 리해성과 이식성과의 차이가 심한것으로 하여 쏘프트웨어개발규모가 확대됨에 따라 점차 FORTRAN, C 등과 같은 수속지향프로그람으로 바뀌였다.

수속지향프로그람은 수속지향적인 문제해결방법에 따르며 그의 기본목적은 콤퓨터가 능히 리해할수 있는 론리를 리용하여 해결할 문제와 그의 구체적인 해결과정을 묘사하고 표현하는것이다. 수속지향문제풀이에서 기본은 자료구조와 알고리듬이다. 여기서 **자료구조**는 콤퓨터의 리산론리를 리용하여 해결하려는 문제를 량자화하여 표현한것이며 **알고리듬**은 문제해결의 구체적인 과정을 어떻게 빨리 효과적으로 진행할수 있는가를 표현한것이다. 수속지향의 문제풀이수법은 구체적인 풀이과정(여기에서 과정은 보통 조작을 가리킨다.)을 정확하게 묘사할수 있으나 호상 련관이 있는 과정들이 얽히여있는 복잡한 체계를 정확하게 표현하기는 어렵다. 반면에 객체지향의 문제풀이는 이러한 문제점을 능히 담당하여 처리할수 있다.

객체지향문제풀이는 고립적인 단일한 과정이 아니라 이 모든 과정들을 표현하는 모체체계에 기초한다. 이것은 콤퓨터론리를 리용하여 체계자체를 모의할수 있게 하며 여기에는 체계의 구성과 체계의 각종 상태, 체계에서 나타날수 있는 각종 과정과 그를 일으키는 체계상태의 절환 등이 포함된다. 객체지향기술은 완전히 새로운 프로그

람설계사상과 관찰, 표현, 문제처리의 방법을 가지고있다. 이러한 객체지향프로그람의 설계와 문제해결능력은 사람들의 일상적이고 자연적인 사유습관과 부합된다.

최초의 객체지향쏘프트웨어는 1966년에 제출된 Simula I로서 여기에서 인간을 모의하는 사유방법을 제기하고 자료와 그와 련관있는 조작들을 하나로 모으려는 사상을 내놓았다. 그러나 당시 하드웨어조건의 제한성과 방법자체의 불충분한 성숙으로 인하여 이 기술을 널리 사용할수 없었다. 그후 쏘프트웨어위기가 출현하고 수속화개발방법의 제한성이 뚜렷해지면서 객체지향방법으로 다시 전환하기 시작하였다. 1980년에 나온 Smalltalk-80언어는 비교적 성숙된 쓸모있는 객체지향도구로서 일련의 객체지향적인 응용을 확고히 실현하고있었다. 이 언어에서 새로운 사상과 문제해결의 새로운 방도, 새로운 방법이 제시되였고 사람들에게 비록 낯설어도 객체지향이라는 개념의 전망적인 발전방향을 보여주었다. 그후 Lisp, Classal, Object pascal, C++ 등 많은 객체지향언어가 나왔으며 그 가운데서도 객체지향기술에 대한 보급추동력이 가장 큰것은 C++이였다.

C++언어는 C언어의 기반우에서 객체지향의 련관내용과 규칙들을 추가한것이다. 많은 문법규칙이 C언어와 류사하므로 C프로그람작성자가 받아들이기 쉽고 동시에 C++가 가지고있는 객체지향기능은 응용프로그람의 개발, 설계와 유지를 간단화하였으므로 큰 규모의 프로그람을 개발하는데 많은 편의를 제공하여 주었다. C++의 광범한 보급과 성공적인 응용은 새로운 객체지향기술의 실력과 전망을 실증하여 주었으며 C++는 C를 대신하는 기본주류의 프로그람작성언어로 되였다.

Java는 90년대 새로 출현한 객체지향프로그람작성언어로서 C++와 류사하지만 C++에서 C언어부분과 비객체지향적인 내용들을 빼버리고 Smalltalk의 완전한 객체지향성의 사상과 많은 사람들에게 습관된 C++의 명령형식을 계승하여 나왔다. Java는 한번 작성하여 여러 실행환경에서 사용하는것을 목적으로 하고있다. 객체지향프로그람설계방법의 출현과 광범한 응용은 콤퓨터쏘프트웨어의 기술발전에서 하나의 중대한 변혁으로, 비약으로 되였다.

1.1.2. 객체와 클라스

객체에 대한 개념은 객체지향기술에서 중심을 이룬다. 객체지향관점에서 보면 모든 객체지향프로그람은 객체로 구성된다. 이 객체들은 자료와 조작이 내장된것으로서 서로 통신, 협조, 배합하여 프로그람의 과제와 기능을 완성한다. 객체지향기술에서의 **객체(Object)**는 현실세계의 어떤 구체적인 물리적인 실체를 콤퓨터론리에로 옮긴것이다. 실례로 텔레비존수상기는 구체적인 존재물로서 외형, 크기, 색갈 등 외적속성과 켜기, 끄기, 통로선택 등 실제적인 기능들을 포함하고있다. 이러한 실체는 객체지향프로그람에서 콤퓨터가 리해할수 있고 조종할수 있으며 어떤 속성과 행동을 가지는 객체로 표현할수 있다.

클라스도 객체지향기술에서 아주 중요한 개념이다. 간단히 말하여 **클라스(Class)**는 같은 종류의 객체들의 모임과 추상이다. 실례로 일상생활에서 보게 되는 휴대형

TV나 천연색TV들은 TV의 범주에 속하며 이 실체들은 객체지향프로그람에서 서로 다른 객체로 넘겨질것이다. 이 서로 다른 TV실체를 나타내는 객체들사이에는 실제적으로 많은 공통성이 있다. 실례로 이것들은 모두 TV신호를 접수하고 방영할수 있으며 화면효과, 음향을 조절할수 있다. 그러므로 문제처리의 편의를 보장하기 위하여 객체지향의 프로그람설계에서는 클라스의 개념을 정의하여 동일한 객체의 공통적속성을 표현하고있다. 이러한 의미에서 클라스는 추상적인 자료형이며 일정한 공통성을 가지고있는 모든 객체들의 추상이다.

클라스에 속하는 매개 객체들을 **클라스의 실체**라고 하며 클라스의 실례화의 결과라고도 한다. 클라스와 객체사이의 관계는 현실세계에서 쉽게 리해할수 있다. 만일 클라스가 추상적인 개념 례를 들어 《TV》이라면 그의 객체는 바로 어떤 구체적인 TV 즉 《1983년에 생산한 아리랑상표 천연색TV》이다.

그림 1-1에서는 클라스, 객체, 실체의 호상관계와 객체지향문제풀이의 사유방식을 보여주고있다.

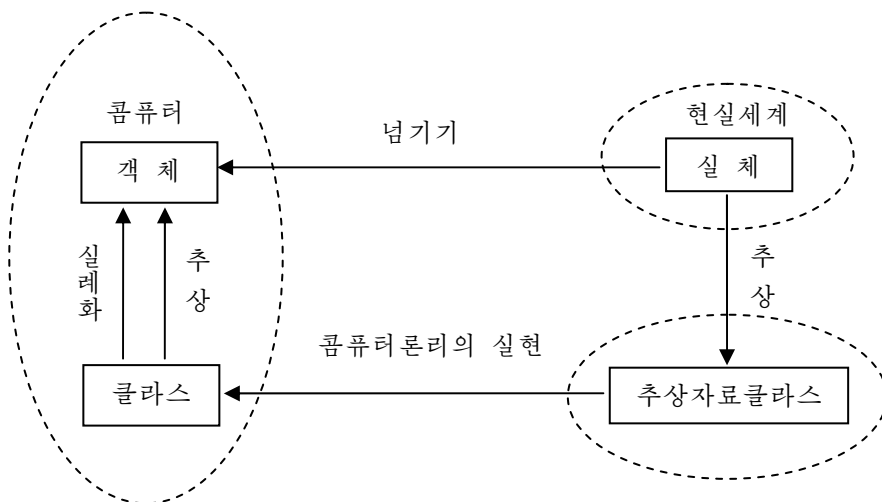



그림 1-1. 객체, 실체와 클라스

객체지향방법을 리용하여 현실세계의 문제를 해결할 때 우선 물리적존재인 실체를 개념세계의 추상자료형으로 추상화한다. 이 추상자료형안에는 실체에서 해결하여야 할 문제와 련관이 있는 자료와 속성들이 포함된다. 다음으로 객체지향도구인 Java언어를 리용하여 이 추상자료형들을 콤퓨터론리를 써서 표현한다. 즉 콤퓨터가 능히 리해하고 처리할수 있는 클라스를 구성한다. 마지막에 클라스를 실례화하여 현실세계의 실체를 객체로 넘긴다. 이렇게 프로그람에서는 객체에 대한 조작을 진행하여 현실세계의 실체에 대한 문제를 모의하고 해결할수 있다.

사실상 객체지향기술의 설계사상은 현실세계의 물리적존재를 콤퓨터론리로 모형화할것을 요구한다. 이렇게 하여 콤퓨터세계를 현실세계에 접근시키게 된다. 이 점은

전통적인 프로그람설계에서 현실세계의 문제를 콤퓨터가 리해하고 처리할수 있는 자료구조로 추상화하는 사고방향 즉 현실세계가 콤퓨터세계에 접근하는 사고방향과 완전히 다르다. 객체지향기술이 제기하는 이 새로운 문제풀이방향은 사람의 사유모형에 보다 접근하고 현실문제에 더 접근하는 방법으로 풀이모형을 설계할수 있게 한다.

제2절. 객체의 속성과 호상관계

- **상태와 행위는 객체의 주요속성이다.**
- **객체들사이의 관계는 3가지 즉 포함, 계승, 련관이다.**

1.2.1. 객체의 속성

상태와 행위는 객체의 주요속성이다.

상태는 객체의 정적속성이라고도 하는데 주로 객체내부가 포함하는 각종 정보를 가리킨다. 다시말하여 변수이다. 매개 객체는 자체의 내부변수를 가지며 이 변수들의 값은 객체가 처한 상태를 나타낸다. 객체가 어떤 조작과 행위에 의하여 상태변경을 일으킬 때 그에 대한 내용적변경을 구체적으로 표현한다. 실례로 TV가 가지고있는 상태정보 즉 종류, 상표, 외관, 크기, 색갈, 개폐기, 통로 등은 콤퓨터에서 변수를 리용하여 표시할수 있다.

행위는 객체의 두번째속성으로서 객체의 메쏘드를 의미한다. 이것을 객체의 동적속성이라고도 하는데 객체의 상태를 설정하거나 변경하는 작용을 한다. 실례로 TV는 켜기, 끄기, 음량조절, 밝기조절, 통로변경 등 조작을 진행할수 있다. 이때 객체의 조작은 일반적으로 객체내부의 변수에 기초하여 이 변수들의 값을 변경하게 된다.(즉 객체의 상태를 변경한다.) 실례로 《켜기》의 조작은 오직 끄기상태에 있는 TV를 유효하게 하며 《켜기》의 조작실행후에는 TV의 끄기상태는 켜기상태로 변할것이다.

1.2.2. 객체들사이의 관계

복잡한 체계는 반드시 많은 객체를 포함하게 된다. 이때 이 객체들사이에는 3가지 관계 즉 포함, 계승, 련관이 있다.

1) 포함

객체 A가 객체 B의 속성일 때 객체 B는 객체 A를 **포함**한다고 말한다. 실례로 매 TV는 수상관을 가지고있다. 수상관을 콤퓨터론리의 객체로 추상화하는 경우 그것과 TV객체사이는 포함관계이다. 하나의 객체가 다른 객체를 포함할 때 그것은 자기의 기억공간안에 포함객체를 위한 전문적인 공간을 남겨야 한다. 즉 포함객체는 그것

을 포함하는 객체내부에 존재하게 된다. 이것은 수상관이 TV에 포함된다는것과 같으며 수상관이 TV의 구성부분이라는것을 의미한다.

2) 계승

객체 A가 객체 B의 특수경우일 때 객체 A는 객체 B를 **계승**했다고 한다. 실례로 흑색TV는 TV의 일종의 특수경우이며 천연색TV는 TV의 다른 특수경우이다. 만일 흑색TV와 천연색TV를 각각 흑색TV객체와 천연색TV객체로 추상화하면 이 객체들과 TV객체사이는 계승관계이다.

객체사이의 계승관계는 바로 클라스사이의 계승관계이다. 특수경우로 되는 클라스를 **하위클라스**라고 하며 하위클라스가 계승하는 클라스를 **상위클라스**라고 한다. 상위클라스는 하위클라스들의 공통관계의 모임이며 하위클라스는 상위클라스에서 정의한 공통속성에 기초하면서 자기의 특수성에 근거하여 자기의 속성을 특별히 정의한다. 실례로 천연색TV객체는 TV객체가 가지는 모든 속성외에 정적속성인 《색도》와 동적속성인 《색도조절》을 특별히 정의한다.

3) 련관

객체 A의 인용이 객체 B의 속성일 때 객체 A와 B사이는 **련관관계**라고 한다. **객체의 인용**이라는것은 객체를 가리키는 명칭이나 주소 등으로서 이 객체를 얻고나 조정할수 있는 경로이다. 객체자체에 비하여 객체의 인용이 차지하고있는 기억공간은 될수록 적어야 하며 그것은 단지 객체를 찾는 실마리일뿐이다. 그것을 통하여 프로그람은 정확한 객체를 찾을수 있으며 객체의 자료에 접근하고 객체의 메쏘드를 호출할수 있다. 실례로 매개 TV는 하나의 생산공장에 대응하며 만일 생산공장을 공장객체로 추상화하면 TV객체는 자기의 생산공장이 어데인가를 기록하여야 한다. 이때 TV객체와 공장객체사이는 련관관계이다.

련관과 포함은 서로 다른 관계이다. 공장은 TV의 구성부분이 아니므로 TV객체는 공장객체 전체를 의미하지 않으며 공장객체의 인용만을 보존한다.(례: 공장의 명칭) 이렇게 하여 공장객체를 요구할 때 즉 공장으로부터 부속품을 구매할것을 요구할 때 TV객체에 보존한 공장이름에 근거하여 이 공장객체를 쉽게 찾을수 있다.

제3절. Java Application프로그람

- Java는 해석형의 프로그람작성언어이다.
- Java Application, Java Applet가 있다.
- Java Application은 몇개의 클라스들과 main메쏘드로 구성된 독자적인 응용프로그람이다.
- Java프로그람작성과정은 3단계 즉 원천코드의 편집, 바이트코드에로의 번역, 바이트코드의 실행이다.

Java언어는 망프로그람작성언어로서 언어의 객체지향, 교차기반, 분산응용 등의 특징은 프로그람작성자들에게 참신한 계산개념을 가져왔다. 또한 WWW가 최초의 단순한 정적봉사로부터 현재의 다양한 동적봉사로 발전하는데서 커다란 변화를 가져왔다.

Java는 소규모응용프로그람을 작성하여 망페지에 삽입하는 음성 및 동화상기능을 실현할수 있을뿐아니라 독점적인 대중규모응용프로그람에도 응용할수 있다. 이 강력한 망기능은 인터네트전체를 하나의 통일적인 실행기반으로 되게 하였다. 구조와 실행환경이 같지 않다는데로부터 Java프로그람은 2가지 류형 즉 Java Application과 Java Applet로 나눌수 있다. 간단히 말하면 Java Application은 독자적인 프로그람이며 Java Applet는 HTML에 삽입되여 실행되는 Web망페지환경의 비독자적인 프로그람이다. 즉 Web열람기내부에 포함하고있는 Java해석기에 의하여 해석실행된다.

일반적으로 고급언어프로그람작성은 원천코드의 편집, 목적코드에로의 번역과 실행이라는 몇가지 단계를 걸친다. Java프로그람 역시 원천코드편집과 바이트코드에로의 번역, 바이트코드의 해석실행단계를 가진다. 아래에서 가장 간단한 Java Application프로그람실례를 통하여 3가지 단계를 설명한다.

1.3.1. 원천프로그람편집

Java원천프로그람은 확장자가 java인 본문파일이며 각종 Java통합개발환경의 원천코드편집기를 리용하여 작성할수도 있고 다른 본문편집기를 리용하여 작성할수도 있다.(례: Windows95의 NotePad나 DOS의 EDIT프로그람 등) 아래에 가장 간단한 Java Application의 실례를 보여주었다.

실례 1-1.

```
Example 1-1 MyJavaApplication
1: import java.io.*;
2: public class MyJavaApplication
3: {
4:      public static void main(String args[])
5:      {
6:           System.out.println("Hello, Java World!");
7:      } // main메쏘드의 끝
8: } // 클라스의 끝
```

프로그람설명

실례 1-1에서 앞에 있는 행번호들은 해석을 편리하게 하기 위하여 붙인것이며 Java프로그람에서는 이 번호를 붙이지 않는다. 1행에서는 import명령문을 리용하여 이미 정의한 클라스들을 이 프로그람에 인입시켜 사용하는데 이것은 C프로그람에서 #include문을 리용하여 서고함수를 적재하는것과 류사하다. 2행의 예약어 class는 클라스정의의 시작을 의미한다. 클라스는 클라스머리부분(2행)과 클라스본체부분(3-8행)으로 정의한다. 클라스본체부분의 내용은 대괄호로 묶어주며 클라스본체내부에서는 다른 클라스를 재정의할수 없다.

임의의 Java프로그람은 몇개의 이러한 클라스정의들로 구성되는데 이것은 C프로그람이 몇개의 함수로 구성되는것과 비슷하다. 주의해야 할것은 Java는 대소문자를 구별하는 언어이므로 class와 CLASS는 서로 다른 정의를 의미한다. 클라스정의는 반드시 예약어 class를 사용해야 한다. 실례에서는 한개의 클라스만을 정의하고 클라스이름을 MyJavaApplication으로 하였다.

클라스본체는 보통 2가지 구성성분을 가진다. 하나는 **마당**으로서 변수, 상수, 객체배렬 등 독립적인 실체들을 포함하며 다른 하나는 **메쏘드**로서 함수와 류사한데 이 두 구성성분을 보통 **클라스의 성원**이라고 한다. 우의 실례에서 MyJavaApplication클라스에는 한개의 클라스성원 즉 main메쏘드만이 있다. 우의 실례의 4행이 main메쏘드의 머리부정의이며 5-7행은 main메쏘드의 본체부분이다. 메쏘드머리부를 표식하는데는 한쌍의 소괄호가 쓰이며 소괄호앞에 있는것이 메쏘드이름이다.(례: main, run, handleEvent 등) 소괄호안에 있는것은 이 메쏘드가 사용하는 형식파라메터이며 메쏘드이름앞에 있는것은 이 메쏘드속성을 설명하는데 쓰이는 장식부이다.

구체적인 문법은 뒤에서 소개한다. 메쏘드본체부분은 반두점(;)들로 구분되는 몇개의 명령문들로 이루어지고 한쌍의 대괄호로 묶으며 메쏘드본체내부에서는 다른 메쏘드를 재정의할수 없다.

main메쏘드는 특수한 메쏘드로서 모든 Java Application의 입력점이다. 그러므로 임의의 Java Application은 반드시 main메쏘드를 가져야 한다. 그리고 main메쏘드의 머리부는 반드시 아래의 형식에 따라 씌여져야 한다.

```
public static void main(String args[])
```

Java Application을 실행할 때 모든 프로그람은 main메쏘드본체의 첫번째 명령문으로부터 실행을 시작한다. 우의 실례에서 main메쏘드는 다음과 같은 한개의 명령문만을 가지고있다.

```
System.out.println("Hello, Java World!");
```

이 명령문은 문자렬 《Hello, Java World!》를 체계의 표준출구(례: 화면)로 출력한다. 여기서 System은 체계의 내부가 정의하는 체계객체이다.

out는 System객체의 마당이며 역시 객체이다. println는 out객체의 메쏘드로서 체계의 표준출구로 지정한 문자렬을 출력한다.

본문편집기를 리용하여 우에서 서술한 프로그람을 콤퓨터에 입력하며 이름이 MyJavaApplication.java인 원천파일로 보관하고 다음 단계 즉 원천코드의 번역에로 들어간다.

1.3.2. 바이트코드에로의 번역

고급언어프로그람의 원천코드로부터 목적코드에로의 생성과정을 **번역**이라고 한다. Java프로그람에서는 원천코드의 번역을 통하여 얻은 목적코드를 **바이트코드**라고 한다. 바이트코드는 2진파일이며 프로그람작성자는 그것을 직접 읽어볼수 없고 Java언어의 해석기로 바이트코드를 해석집행한다. 바이트코드에로의 번역은 전용 Java번역기를 사용하여 진행하며 통합 Java개발환경(Sun One Studio, Jbuilder, Visual J++ 등)에서 차림표명령이나 어떤 단추를 찰칵하여 번역과정을 완성할수도 있다.

실례 1-1의 원천코드를 바이트코드로 생성하자면 아래의 명령을 실행해야 한다.

```
Javac MyJavaApplication.java
```

이 명령을 리용하면 Java번역프로그람 javac.exe를 호출하여 원천코드파일 MyJavaApplication.java에 문법오유가 있는가를 검사한 다음 상응한 바이트코드파일을 생성한다. 주의할것은 원천프로그람이름이 완전히 주어져야 하며 대소문자의 정확성을 지켜야 한다는것이다. 그렇지 않으면 번역오유를 발생시킬수 있다.

C언어 등의 다른 고급프로그람언어의 번역은 보통 한개의 완전한 코드파일을 한개의 목적코드파일로 생성하지만 Java프로그람의 번역은 원천코드파일에서 정의한 매개 클라스에 대응하여 이 클라스이름에 class확장자가 붙은 바이트코드파일을 생성한다.

그러면 2개의 클라스를 정의한 Java프로그람실례를 고찰하자.

실례 1-2

```
Example 1-2 MyApplication2.java
1: import java.io.*;
2: public class MyApplication2
3: {
4:      public static void main(String args[])
5:      {
6:           System.out.println(UserClass.m_sMessage);
7:      }
8: }
9:
10:class UserClass
11:{
12:     static String m_sMessage = "Message from User Defined Class";
13:}
```

프로그람설명

이 실례에서는 2개의 클라스를 정의하고있는데 하나는 main메쏘드의 주클라스 MyApplication2이며 다른 하나는 마당 m_sMessage를 가진 클라스 UserClass이다. m_sMessage는 문자렬객체이며 클라스를 정의할 때 그것에 초기값을 주고있다. 주클라스 MyApplication2의 main메쏘드는 이 문자렬객체를 리용하여 그의 초기값을 화면에 출력한다. 주의할것은 Java원천코드파일에서 여러개의 클라스들을 정의할수 있으나 여기서 하나의 클라스만이 main메쏘드를 가질수 있다는것이다. main메쏘드는 Java Application프로그람집행의 입력점이며 main메쏘드를 포함하는 클라스를 주클라스라고 한다. 주클라스이름을 Java원천코드파일이름으로 한다.

아래의 명령을 사용하여 실례 1-2의 원천코드를 번역하면 2개의 바이트코드파일 MyApplication2.class와 UserClass.class를 얻을수 있다.

```
Javac MyApplication2.java
```

1.3.3. 바이트코드의 해석과 실행

그림 1-2에서와 같이 고급프로그람작성언어는 실행방식에 따라 번역형과 해석형으로 구분할수 있다. 번역형언어는 C, Pascal 등이며 이것이 생성하는 목적코드는 련결(link)후에 직접 실행할수 있는 실행가능코드로 된다. 한편 해석형언어는 Basic, Java 등이며 이 프로그람은 조작체계준위에서 직접 실행할수 없고 전문적인 해석프로그람이 있어서 해석집행하여야 한다.

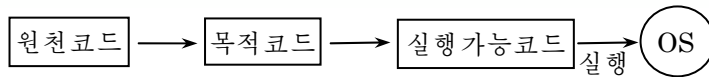
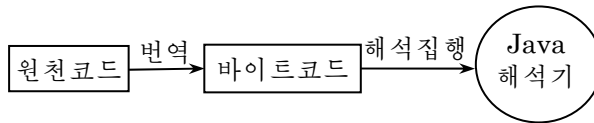



그림 1-2. 전통적인 언어와 Java의 실행기구

일반적으로 해석형언어는 비교적 간단하나 실행속도가 느리다. 그러나 망응용기반에서 해석형언어는 오히려 기본적인 우세를 보이고있다. 번역형언어는 조작체계와 직접적인 련관이 있으므로 그것을 실행시키는 쏘프트하드웨어기반에 비교적 강하게 의존한다.

어떤 기반상에서 정상적으로 실행할수 있는 번역프로그람이 다른 기반상에서는 동작할수 없으며 그러므로 이 기반에서 반드시 원천코드를 다시 번역하여 이 기반에 적합한 실행가능코드를 생성하여야 한다. 이러한 이식성에서의 부족점은 기반을 지원하는 망응용프로그람에 있어서 매우 불편한것으로 된다. 망은 서로 다른 쏘프트하드웨어기반의 콤퓨터들로 구성되였기때문에 번역형응용프로그람을 오유없이 실행할수 있게 하기 위하여서는 서로 다른 기반에 대하여 반드시 서로 다른 판본의 응용프로그람을 전문적으로 개발해야 하며 그와 동시에 판본상승과 유지보수에 대한 작업량도 매우 커지게 된다.

해석형언어는 이 문제를 해결하기 위하여 완전히 새로운 사상을 내놓았는데 Java가 바로 이 사상을 받아들인것이다. Java원천코드번역에 의하여 생성된 바이트코드는 일반적인 조작체계기반에서 직접 실행할수 없으며 반드시 조작체계밖의 《Java 가상기계》라는 쏘프트웨어체계우에서 실행하여야 한다. Java프로그람을 실행할 때 먼저 이 가상기계를 기동하고 다음에 Java바이트코드를 해석집행해야 한다. 이렇게 Java 가상기계를 리용하면 Java바이트코드를 쏘프트웨어체계에 따라 구분할수 있으며 서로 다른 콤퓨터에서 이 구체적인 체계특성에 따라서만 Java가상기계를 설치하여야 할뿐이다. 이것은 서로 다른 쏘프트하드웨어기반의 구체적인 차이점을 감출수 있게 한다.

Java Application은 독자적인 해석프로그람으로 실행하며 해석프로그람은 java.exe이다. 실례 1-1에서 생성한 MyJavaApplication.class는 아래의 명령문을 사용하여 실행할수 있다. 즉

```
java MyJavaApplication
```

실행결과 화면상에 다음과 같이 현시된다.

```
Hello, Java World!
```

실례 1-2도 같은 방법으로 실행할수 있다.

```
java MyApplication2
```

실행결과는 다음과 같다.

```
Message from User Defined Class
```

제4절. Java Applet프로그람

- Java Applet는 HTML파일에 삽입하여 실행하는 Web망페지환경의 비독자적인 프로그람이다.
- Java Applet의 어느한 클라스는 Applet클라스의 하위클라스가 되여야 한다.

Java Applet는 다른 류형의 중요한 Java프로그람이며 그의 원천코드편집과 바이트코드의 번역과정은 Java Application과 같지만 독자적으로 실행할수 있는 프로그람이 아니다. Java Applet의 바이트코드파일은 반드시 HTML의 파일에 삽입하여 HTML파일을 담당해석하는 WWW열람기에 의해 해석집행하여야 한다. HTML은 인터네트상에서 가장 광범히 응용되는 통용언어로서 망상에서 서로 다른 다매체정보를 WWW열람기에 올릴수 있게 한다. 한편 Java Applet는 HTML의 정보내용과 표현방식을 보다 풍부하게 한다. Java언어가 출현한 초기에 처음으로 리용한것이 Java Applet이다.

1.4.1. 원천코드의 편집과 번역

가장 간단한 Java Applet프로그람을 실례들어 보자.

실례 1-3

```
Example 1-3 MyJavaApplet.java
1: import java.awt.Graphics;
        //java.awt패키지의 체계클라스 Graphics를 프로그람에 인입
2: import java.applet.Applet;
        //java.applet패키지의 체계클라스 Applet를 프로그람에 인입
3: public class MyJavaApplet extends Applet
4: {
5:     public void paint(Graphics g)
6:     {
7:         g.drawString("Hello, Java Applet World!", 10, 20);
8:     } //end of paint method
9: } //end of class
```

프로그람설명

이 프로그람에서는 주석행표시 《//》을 사용하였다. 기호 《//》다음의 모든 내용은 프로그람에 대한 설명이며 번역기와 해석기에서 무시된다.

우선 프로그람의 1, 2행은 import예약어를 리용하여 프로그람이 리용하여야 할 체계클라스 Applet와 Graphics를 인입하고있다. 이 체계클라스들은 각각 서로 다른 체계패키지에 위치하고있으며 인용시 그것들이 있는 패키지이름을 지적하여야 한다. 패키지(package)는 Java체계가 체계클라스를 조직하는데 쓰이는 배렬로서 기능에 있어서 련관있는 클라스들을 묶어놓은것이다.

실례 1-3의 클라스이름은 MyJavaApplet(3행)이다. Java Applet프로그람 역시 몇개의 클라스들의 정의로 구성된다. 클라스정의는 class예약어에 의해 표시된다. 그러나 Java Applet에서는 main메쏘드를 가지지 않는다. 중요한것은 프로그람에 체계클라스 Applet의 하위클라스가 반드시 있어야 한다는것이다. 다시말하여 클라스머리부분에 extends Applet꼬리부가 반드시 있어야 한다는것이다. 여기서 extends는 예약어로서 새로 정의하는 클라스가 그 뒤에 있는 클라스의 하위클라스라는것을 의미한다. Applet는 상위클라스이름이며 그것은 체계클라스일수도 있고 기타 다른 사용자정의클라스일수도 있다. 하위클라스는 상위클라스로부터 일부 성원을 계승한다. 즉 마당과 메쏘드를 계승한다.

모든 Java Applet프로그람에는 반드시 체계클라스 Applet의 하위클라스가 있어야 한다. 사용자프로그람에서 Applet의 하위클라스는 상위클라스의 련관성원들을 자동적으로 사용할것이며 WWW열람기가 사용자프로그람이 정의하는 작업들을 문제없이 실행하게 한다.

실례 1-3의 4-9행은 클라스 MyJavaApplet의 본체부분이다. 여기서는 한개의 메쏘드 paint만을 정의하였다. 사실상 paint메쏘드도 체계클라스 Applet에서 이미 정의한 메쏘드이다. 그러므로 사용자프로그람이 정의하는 Applet하위클라스는 이 메쏘드를 계승하며 구체적인 요구에 따라 그 내용을 고쳐써서(이 과정을《다중정의》라고 한다.) WWW열람기가 Java Applet프로그람을 해석할 때 이 성원메쏘드를 자동실행하여 (례: paint메쏘드) 필요한 기능을 실현한다.

paint메쏘드는 WWW가 현시하는 Web홈페지를 그려야 할 때(례: 열람기창문을 화면상에서 이동, 확대, 축소할 때) 열람기에 의해 자동적으로 호출되며 일반적으로 열람기에서 외적인 대면부를 그려낸다. 열람기는 Applet프로그람이 있는 HTML파일을 열람할 때 적당한 시각에 이 paint메쏘드를 자동실행하여 화면상에 현시하려는 정보를 출력한다.

실례 1-3의 paint메쏘드는 아래의 명령문을 리용하여 화면의 지정한 위치에 문자렬 《Hello, Java Applet World!》를 출력한다.

```
g.drawString("Hello, Java Applet World!", 10, 20)
```

여기서 g는 체계클라스 Graphics의 객체이다. 그것은 Web페지에서 Applet프로그람대면부의 배경을 나타내고있으며 g의 메쏘드를 사용하여 문자렬을 현시하면 Applet프로그람대면부에 문자렬이 표시된다.

Java Application과 Java Applet는 실행방식에서 큰 차이를 가지고있지만 그것들은 같은 Java언어의 문법규칙에 따르며 같은 번역도구를 사용한다. 만일 실례 1-3을 번역하려면 아래의 명령문을 사용할수 있다.

javac MyJavaApplet.java

번역한 결과 현재등록부에 원천코드의 클라스이름 MyJavaApplet로 지정한 바이트코드파일 MyJavaApplet.class가 생성된다.

1.4.2. 코드삽입

Java Applet를 실행할 때 반드시 이 바이트코드를 HTML파일에 삽입해야 한다.

실례 1-3의 Java Applet프로그람을 다음과 같이 HTML파일에 삽입할수 있다.

실례 1-4

```
Example 1-4 AppletInclude.html
1: <HTML>
2:     <BODY>
3:         <APPLET CODE = "MyJavaApplet.class" HEIGHT = 200 WIDTH = 30>
4:         </APPLET>
5:     </BODY>
6: </HTML>
```

프로그람설명

HTML은 여러가지 꼬리표를 가지고 하이퍼본문정보를 편집배렬한다. <HTML>과 </HTML>꼬리표는 HTML파일의 시작과 마감을 의미한다. HTML에서 Java Applet의 삽입은 약속한 특수꼬리표 〈APPLET〉와 〈/APPLET〉를 리용하여 진행한다. 여기서 〈APPLET〉꼬리표는 3개의 파라메터를 가지고있다.

- CODE: HTML파일에 삽입하려는 JavaApplet바이트코드파일이름을 지정한다.
- HEIGHT: Web페지에서 차지하는 구역의 높이를 지정한다.
- WIDTH: Web페지에서 차지하는 구역의 너비를 지정한다.

보는바와 같이 Java Applet바이트코드를 HTML파일에 삽입할 때 실제상 바이트코드파일의 파일이름만 삽입하였을뿐이다. 실지 바이트코드파일 자체는 HTML파일과 같은 경로에 독립적으로 보존되며 WWW열람기는 HTML파일에 삽입한 이름에 근거하여 이 바이트코드파일을 자동탐색하고 실행한다.

1.4.3. Applet의 실행

Applet의 실행과정은 그림 1-3에서 보여주었다.

우선 번역한 바이트코드파일과 작성한 HTML파일(여기에 바이트코드파일이름을 포함)을 Web봉사기의 적당한 경로에 놓이게 한다. WWW열람기가 HTML파일을 내리적재하여 현시할 때에 HTML이 지정한 Java Applet바이트코드를 자동적으로 내리적재하며 다음에 Java해석기를 리용하여 이 기계에 내리적재한 바이트코드프로그람을 해석집행한다.

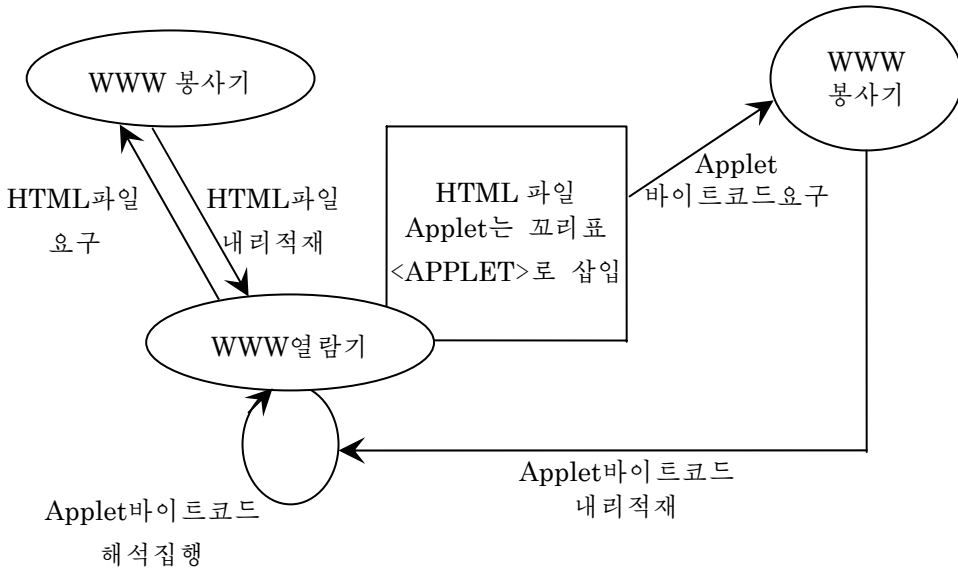



그림 1－3. Java Applet의 내리적재집행과정

이 과정에서 볼수 있는바와 같이 Java Applet의 바이트코드프로그람은 처음에는 Web봉사기상에 존재한다. 실행과정은 이 지점에서 내리적재한 후에 자기의 거점기계에서 완료된다. Applet프로그람을 수정하거나 보수하여야 하는 경우 봉사기측의 프로그람을 교체 실행하기만 하면 된다.

Java해석기를 내장한 Web열람기(례: 3.0판이상의 IE 혹은 Netscape Navigator)를 선택하여 실례 1-4의 AppletInclude.html파일을 열면 JavaApplet의 실행결과를 볼수 있다.

그림 1-4는 실례 1-4의 실행결과이다.

JDK쏘프트웨어패키지에서는 WWW열람기를 모의하여 Applet를 실행하는 응용프로그람 AppletViewer.exe를 제공하고있다. 이것을 사용하면 열람기를 반복호출할 필요가 없게 된다. 실례 1-3의 Java Applet프로그람은 아래의 명령을 사용하여 할수 실행할수 있다.

```
appletviewer AppletInclude.html
```

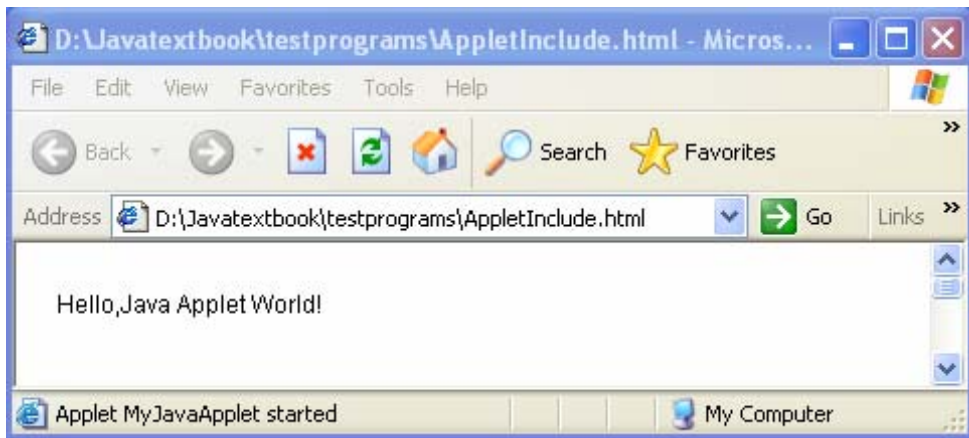



그림 1-4. 열람기에서 Java Applet의 실행결과

제5절. 도형사용자대면부의 입출력

도형사용자대면부의 프로그람작성에서는 반드시 java.awt패키지를 적재.

입출력은 프로그람의 기본기능이다. 5, 6절에서 기본입출력기능을 가지는 Java프로그람을 어떻게 작성하는가를 서술한다. 이 절에서는 우선 GUI를 통한 입출력을 소개한다. **도형사용자대면부(Graphics User Interface)** 간단히 GUI는 대부분 응용프로그람들에서 사용하는 입출력대면부이다. 이것은 도형방식으로 동작하며 조작이 간편하고 리해하기 쉬운 우점을 가지고있다.

1.5.1. Java Applet도형사용자대면부의 입출력

Java Applet프로그람은 WWW열람기에서 실행되며 열람기자체가 도형사용자대면부환경이다. 즉 Java Applet프로그람은 도형사용자대면부에서만 동작할수 있다.

실례 1-5

```
Example 1-5 AppletInOut.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4: public class AppletInOut extends Applet implements ActionListener
```

```
5: {
6:      Label prompt;
7:      TextField input, output;
8:
9:      public void init( )
10:     {
11:         prompt = new Label("Please input your name:");
12:         input = new TextField(6);
13:         output = new TextField(20);
14:         add(prompt);
15:         add(input);
16:         add(output);
17:         input.addActionListener(this);
18:     }
19:     public void actionPerformed(ActionEvent e)
20:     {
21:         output.setText(input.getText( ) + ", Welcome You!");
22:     }
23:}
```

프로그람설명

실례 1-5는 사용자가 입력하는 이름문자렬을 접수하고 사용자가 넣기건(Enter건)을 누를 때 이 문자렬과 새로운 문자렬을 조합하여 출구하는 프로그람이다.(그림 1-5)

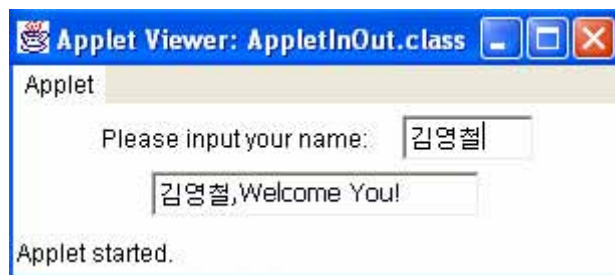



그림 1-5. 실례 1-5의 실행결과

프로그람의 1-3행은 각각 Java클라스서고의 3개의 패키지(java.applet.*, java.awt.*, java.awt.event.*)를 적재한다. Java Applet프로그람이므로 반드시 java.applet패키지를 적재하여야 하며 도형사용자대면부를 사용하려면 java.awt패키지를 적재하여야 한다. 또한 도형사용자대면부의 사건처리를 사용한다면 java.awt.event패키지를 적재하여야 한다. 4행은 클라스 AppletInOut를 정의하였다. Java Applet프

로그람작성규칙에 따라 이 클라스는 Applet클라스의 하위클라스이여야 하며 extends Applet를 사용하여 나타낸다. implements ActionListener는 이 클라스가 동시에 동작사건(ActionEvent)의 감시자라는것을 의미한다.

6, 7행은 표식자(Label)객체인 prompt와 2개의 본문마당(TextField)객체인 input와 output를 정의하고있다. 여기에서 prompt는 정보를 제시하는데 쓰이고 input는 사용자입력정보를 접수하는데 쓰이며 output는 프로그람처리의 결과정보를 출력하는데 쓰인다.

9-18행은 AppletInOut클라스의 init( )메쏘드를 정의하고있으며 public와 void는 init( )메쏘드의 **장식부**이다. init( )메쏘드는 열람기가 Java Applet프로그람을 사용할때 자동적으로 실행된다. 실례에서 이 메쏘드는 6, 7행에서 정의한 객체들을 창조(11-13행)하며 Applet프로그람의 도형사용자대면부에 추가(14-16행)한다. 17행에서 input객체를 Action사건의 감시자에게 등록한다. 그렇게 하지 않으면 사용자가 input에서 Enter건을 누르는 조작에 프로그람은 응답할수 없다. 19-22행은 AppletInOut클라스의 다른 메쏘드인 actionPerformed( )를 정의하고있으며 동작사건의 감시자는 이 메쏘드를 사용하여 동작사건을 처리한다. 실례에서는 입력칸에서 Enter건을 누르기만 하면 동작사건이 일어난다. 그러므로 동작사건의 원인을 다시 판단하지 않고 input.getText( )를 직접 리용하여 사용자가 input본문마당에 입력한 이름문자렬을 얻을수 있다. 그리고 문자렬 《,Welcome You!》를 붙혀쓰고 output.setText( )메쏘드를 리용하여 결합된 문자렬을 output본문마당에 현시한다.

실례를 통해 알수 있는바와 같이 도형사용자대면부는 기본적으로 표식자객체나 본문마당객체를 리용하여 자료의 입출력을 실현하며 특히 본문마당객체를 리용하여 사용자가 건반으로 입력한 자료를 얻는다.

1.5.2. Java Application도형사용자대면부의 입출력

Java Applet프로그람과 달리 Java Application프로그람은 열람기가 제공하는 도형사용자대면부가 없이 직접 사용할수 있으며 그러므로 우선 자기의 도형사용자대면부를 창조해야 한다.(실례 1-6)

실례 1-6

```
Example 1-6 ApplicationGraphicsInOut.java
1: import java.awt.*;
2: import java.awt.event.*;
3: public class ApplicationGraphicsInOut
4: {
5:      public static void main(String args[])
6:      {
```

```
7:          new FrameInOut( );
8:      }
9: }
10:class FramInOut extends Frame implements ActionListener
11:{
12:     Label prompt;
13:     TextField input, output;
14:
15:     FrameInOut( )
16:     {
17:         super("도형사용자대면부의  Java Application");
18:         prompt = new Label("이름을 입력하십시오:");
19:         input = new TextField(6);
20:         output = new TextField(20);
21:         setLayout(new FlowLayout( ));
22:         add(prompt);
23:         add(input);
24:         add(output);
25:         input.addActionListener(this);
26:         setSize(300, 200);
27:         show( );
28:     }
29:     public void actionPerformed(ActionEvent e)
30:     {
31:         output.setText(input.getText( ) + ", Welcome You!");
32:     }
33:}
```

프로그람설명

실례 1-6에서는 2개의 클라스를 정의하고있는데 FrameInOut클라스는 java.awt 패키지의 창문클라스 Frame의 하위클라스이며 도형사용자대면부를 작성하고 사용하는데 쓰인다. ApplicationGraphicsInOut클라스는 주클라스이며 main( )메쏘드에서 FrameInOut클라스의 객체와 도형사용자대면부의 창문을 창조한다. 이 실례는 실례 1-5와 완전히 같지만 단지 도형사용자대면부와의 련관작업이 FrameInOut클라스에서 완성된다는것이 다르다. 실례 1-5의 init( )메쏘드와 류사하게 FrameInOut클라스의 FrameInOut( )메쏘드도 FrameInOut객체창조시에 자동적으로 호출실행되며 표식자, 본문마당렬을 창조하고 FrameInOut가 만든 도형사용자대면부의 창문에 가입한다.

17행은 창문의 표제를 나타낸다. 21행은 창문에서 표식자, 본문마당 등 객체를 배치하는 배치방안을 설정한다. 26행에서 창문의 크기를 300화소(너비) × 200화소(높이)로 설정한다. 27행에서 창문을 현시한다.

이렇게 창문객체를 리용하여 Java Application프로그람은 열람기의 도움이 없이 도형사용자대면부의 입출력기능을 실현할수 있다. 그런데 실례 1-6의 프로그람을 사용하기에는 아직 부족한것이 있으므로 창문을 닫는 코드를 더 작성하였다.(실례 1-7)

실례 1-7

```
Example 1-7 ApplicationGraphicsInOut2.java
1: import java.awt.*;
2: import java.awt.event.*;
3: public class ApplicationGraphicsInOut2
4: {
5:      public static void main(String args[])
6:      {
7:           new FrameInOut( );
8:      }
9: }
10:class FrameInOut extends Frame implements ActionListener
11:{
12:     Label prompt;
13:     TextField input, output;
14:     Button btn;
15:
16:     FrameInOut( )
17:     {
18:          super("GUI Java Application");
19:          prompt = new Label("이름을 입력하십시오:");
20:          input = new TextField(6);
21:          output = new TextField(20);
22:          btn = new Button("닫기");
23:          setLayout(new FlowLayout( ));
24:          add(prompt);
25:          add(input);
26:          add(output);
27:          add(btn);
28:          input.addActionListener(this);
```

```
29:         btn.addActionListener(this);
30:         setSize(300, 200);
31:         show( );
32:     }
33:     public void actionPerformed(ActionEvent e)
34:     {
35:         if(e.getSource( ) == input)
36:             output.setText(input.getText( ) + ", Welcome to you!");
37:         else
38:         {
39:             dispose( );
40:             System.exit(0);
41:         }
42:     }
43:}
```

프로그람설명

실례 1-7의 14행에서 단추(Button)객체 btn을 추가한다. 22행은 이 단추를 창조하고 표제를 《닫기》로 정의하였다. 27행은 이 단추를 Application프로그람의 도형사용자대면부에 추가하였다. 29행에서 btn객체를 단추를 눌렀을 때 일어나는 동작사건의 감시자에게 등록한다. 실례 1-7에서는 본문마당외에 단추를 눌러도 동작사건을 일으킬수 있게 한다. 동작사건을 처리하는 actionPerformed( )메쏘드에서는 우선 어느것이 동작사건을 일으키는가를 판단한다. 만일 본문마당 input가 동작사건을 일으켰다면 입력한 문자렬을 얻어《, Welcome to you》와 결합하여 출력한다. 그렇지 않으면 단추 btn을 찰칵하였다고 인식하고 창문을 닫으며 프로그람을 결속한다. 여기서 40행은 Java가상기계 JVM에서 탈퇴하여 JVM을 실행하는 조작체계에 돌아오도록 하는 명령문이다. 결과는 그림 1-6에 보여주었다.

그림 1-6. 실례 1-7의 실행결과

제6절. 문자대면부의 입출력

- Java Application만이 문자대면부의 입출력을 실현한다.
- 문자대면부입출력을 실현하자면 java.io패키지를 적재해야 한다.

문자대면부라는것은 문자방식을 가리키는 사용자대면부이다. 문자대면부에서 사용자는 문자렬을 리용하여 프로그람에 명령을 주어 자료를 전송하며 프로그람실행결과 역시 문자형식을 리용하여 표현한다. 도형사용자대면부가 이미 널리 보급되였어도 문자대면부의 응용프로그람을 사용할것을 요구하는 경우도 있다. 실례로 문자대면부의 조작체계나 문자대면부만을 지원하는 말단 등을 들수 있다. 그러므로 이 절에서는 Java프로그람에서 문자대면부입출력의 기본조작을 서술한다.

모든 Java Applet프로그람은 도형사용자대면부의 열람기에서 실행된다. 그러므로 Java Application만이 문자대면부의 입출력을 실현할수 있다.

실례 1-8

```
Example 1-8 ApplicationCharInOut.java
1: import java.io.*;
2:
3: public class ApplicationCharInOut
4: {
5:       public static void main(String args[])
6:       {
7:           char c = ' ';
8:           System.out.print("Enter a character please:");
9:           try {
10:              c = (char)System.in.read( );
11:          }catch(IOException e){ };
12:          System.out.println("You've entered character " + c);
13:      }
14:}
```

프로그람설명

실례 1-8은 우선 사용자가 건반을 통하여 한개 문자를 입력한 다음 이 문자를 사용자에게 보여준다. 7행에서 문자형(char)변수 c를 창조하고 공백문자를 값주기하여

초기값으로 한다. 8행에서는 화면에 정보를 출력한다. 9-11행은 사용자가 입력하는 문자를 접수하는 부분으로서 사용자가 한개의 문자를 입력하고 Enter건을 누른 후에야 입력한 문자를 문자변수 c에 보존하고 다음 명령을 집행해간다. 12행은 c에 보존된 문자를 화면우에 출력한다. 실례 1-8의 실행결과는 그림 1-7과 같다.

실례 1-8의 프로그람은 오직 사용자가 입력한 하나의 문자만을 접수할수 있으며 만일 사용자가 입력한 여러개의 문자(즉 문자렬)들을 접수하려면 실례 1-9과 같이 수정해야 한다.

```
D:\Javatextbook\Test>javac ApplicationCharInOut.java

D:\Javatextbook\Test>java ApplicationCharInOut
Enter a character please:Q
You've entered character Q

D:\Javatextbook\Test>_
```

그림 1-7. 실례 1-8의 실행결과

실례 1-9

```
Example 1-9 ApplicationLineIn.java
1: import java.io.*;
2: public class ApplicationLineIn
3: {
4:      public static void main(String args[])
5:      {
6:          String s = " ";
7:
8:          System.out.print("please enter a string:");
9:          try
10:         {
11:             BufferedReader in =
12:                     new BufferedReader(new InputStreamReader(System.in));
13:             s = in.readLine( );
14:         }catch(IOException e){ }
15:         System.out.println("You've entered string:" + s);
16:     }
17:}
```

프로그람설명

실레 1-9는 java.io패키지의 2개의 입출력에 관한 클라스 BufferedReader와 InputStreamReader를 사용하고있다. 12행의 System.in은 체계기정의 표준입력(즉 건반)을 의미하며 우선 그것을 InputStreamReader클라스의 객체로 변환하고 다음에 BufferedReader클라스의 객체 in으로 변환한다. 원래의 비트입력은 완충문자입력으로 변화된다. 13행은 readLine( )메쏘드를 리용하여 사용자가 건반으로부터 입력한 한 행 문자를 읽어들여 문자렬객체 s에 값주기한다. 15행은 이 문자렬을 화면우에 현시한다. 그림 1-8은 실레 1-9의 실행결과이다.

```
D:\Javatextbook\Test>java ApplicationLineIn
please enter a string:안녕하십니까
You've entered string:안녕하십니까

D:\Javatextbook\Test>
```

그림 1-8. 실례 1-9의 실행결과

제7절. Java언어의 특징

Java언어의 특징: 기반의 무관계성, 객체지향성, 안전성과 안정성, 다중스레드의 지원, 배우기 쉽다.

앞절들의 여러 실례를 통하여 Java언어에 대한 일련의 특징을 서술할수 있다.

간단히 말하면 Java는 망관련기능이 강한 콤퓨터언어로서 특징은 망상에서의 응용프로그람을 개발하는데 특별히 적합하다는것이다. 다른 한편 Java는 현대쏘프트웨어기술의 새로운 성과(객체지향, 다중토막처리(Thread) 등)들을 집중적으로 체현하고있을뿐아니라 충분히 리용하고있다는것이다.

1. 기반의 무관계성

Java의 가장 큰 특징은 모든 콤퓨터환경(기반)에서 실행가능하다는것이다. 그 원리를 그림 1-9에 주었다. Java언어로 작성된 프로그람은 콤파일되여 바이트코드로 변환되고 이 바이트코드는 Java가상기계(JVM: Java Virtual Machine)에서 실행된다. 이전에 프로그람작성을 하던 프로그람작성자라면 이것이 얼마나 큰 장점이며 편리한것인지 알수 있을것이다. Windows에서 개발된 프로그람을 UNIX나 Linux와 같은 기반에서 실행하려면 거의 불가능하거나 대부분의 원천코드를 다시 작성하여야 한다. 그러나 Java로 작성하면 한번 만들어진 프로그람을 어떤 기반에도 상관없이 실행하고 사용할수 있다.

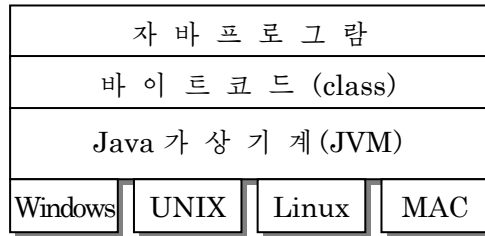



그림 1-9. Java의 독립적인 실행환경

Java가 모든 기반에서 실행가능한것은 Java가상기계가 해당 기반에 설치되여 프로그람작성자에게 모두 같은 실행환경을 제공하기때문이다.

결국 Java를 리용하면 기반에 관계없이 보편적으로 적용되는 응용프로그람을 작성할수 있으며 개발, 유지, 관리의 비용을 크게 저하시킨다.

2. 객체지향

Java는 객체지향의 프로그람작성언어이다. 객체지향기술은 오늘날 쏘프트웨어개발과정에서 새롭게 출현하여 전통적인 수속지향언어에서 처리할수 없는 문제들에 적용되고있으며 쏘프트웨어개발의 규모확대, 성장촉진, 유지량증대 및 개발일정계획의 세분화, 전문화, 표준화 등을 가능하게 하는 쏘프트웨어개발방법이다.

객체지향기술의 핵심은 사람의 사유에 보다 적응한 방법으로서 콤퓨터론리모형이라는것이며 그것은 클라스와 객체의 구조를 리용하여 자료와 조작을 함께 내장하고 통일적인 대면을 통하여 외부와 접촉한다.

현실세계의 실체를 반영하는 매 클라스는 프로그람에서 독립성과 계승성을 가질수 있다. 이러한것은 프로그람의 보수 및 유지와 반복사용성, 개발효률을 높이는데 효과적이며 대규모쏘프트웨어를 쉽게 창조하고 사용유지할수 있게 한다. C++ 역시 객체지향언어이지만 C언어와 겸용해야 하며 여기에 수속지향방법들을 일부 포함하고있다. 그러나 Java는 C++의 수속지향방법을 제거하였으며 프로그람작성은 설계, 클라스실현, 속성정의, 행위의 과정이다.

3. 안전성과 안정성

망상에서의 응용프로그람에 대한 요구는 높은 안정성과 믿음성이다. 사용자는 망을 통하여 정보를 얻으며 실행프로그람은 반드시 믿음성이 있어야 한다. 비루스나 다른 조작을 허용할수 없으며 안정하여야 한다. 또한 쉽게 폭주 등의 오유를 발생하지 말아야 하며 사용자가 사용하기 좋게 하여야 한다. Java는 《모래채》라는 특유한 기구를 가지고있는데 이것은 안전성을 보장한다. 그것은 또한 C++에서 쉽게 오유를 발생시킬수 있는 지적자를 제거하고 자동기억기관리 등의 조작을 추가하고있으며 Java프로그람의 실행믿음성을 보증한다.

4. 다중토막처리의 지원

다중토막처리는 쏘프트웨어기술의 중요한 성과중의 하나이며 조작체계, 응용프로그람개발 등의 여러 령역에서 이미 성공적으로 응용되고있다. 다중토막처리기술은 동일한 프로그람이 두개의 실행통로를 가질수 있게 하며 동시에 두가지 과제를 수행할수 있게 한다. Java는 다중토막처리기능을 가지고있을뿐아니라 언어준위의 다중토막처리지원을 제공하고있으며 다중토막처리를 작성관리하는데 쓰이는 클라스와 메쏘드를 정의하여 다중토막처리기능을 가진 프로그람의 변화를 간단하고 쉽고 유효하게 한다.

5. 쉽게 배울수 있다.

앞에서 서술한바와 같이 C++로부터 파생한 Java언어는 안전성과 안정성을 보장하고 C++에서 쉽게 리해하고 소유할수 없는 부분을 제거하였다. 가장 전형적인것이 지적자조작과 기억기관리기능이다. 한가지 특징을 가지고있는데 그것은 바로 기본언어부분이 C언어와 거의 같다는것이다. 그러므로 Java를 배우고 C언어를 배우든가 이미 C언어를 소유하고 Java를 다시 배우든지간에 쉽다는것을 알수 있다.

Java의 이 특성들은 망응용개발요구에 적응할수 있다.

제2장. Java언어의 기초

이 장에서는 Java프로그람을 작성하고 리해하는데서 반드시 필요한 언어의 기초지식을 주로 소개하며 Java프로그람의 구조, 자료형, 변수, 상수, 표현식과 흐름조종명령문, 묶음, 벡토르, 문자렬 등을 서술한다.

제1절. Java프로그람의 구성방식

Java프로그람은 클라스들의 정의로 구성되며 매 클라스의 내부는 클라스의 정적속성선언과 클라스의 메쏘드 두 부분을 포함한다.

제1장에서 소개한 몇가지 간단한 Java프로그람의 실례를 통하여 Java프로그람의 일반구성규칙을 리해할수 있다. 아래에서 실례 1-5의 프로그람으로 Java프로그람의 구성을 소개한다.

실례

```
Example AppletInOut.java
1: import java.applet. *;
2: import java.awt. *;
3: import java.awt.event.*;
4: public class AppletInOut extends Applet implements ActionListener //주클라스머리부
5: {
6:          Label prompt;                          } 정적속성
7:          TextField input, output;
8:
9:          public void init( )
10:         {
11:                 prompt = new Label("이름을 입력하십시오:");
12:                 input = new TextField(6);
13:                 output = new TextField(20);
14:                 add(prompt);                                  메쏘드 1
15:                 add(input);
16:                 add(output);
17:                 input.addActionListener(this);
18:         }
```

```
19:        public void actionPerformed(ActionEvent e)          ⎫
20:        {                                                    ⎬ 메쏘드 2
21:              output.setText(input.getText( ) + ", 환영합니다!");  ⎭
22:        }
23:}
```

Java원천프로그람은 클라스의 정의들로 구성된다. 그중 한개의 클라스는 주클라스여야 한다. Java Application에서 주클라스는 main메쏘드를 포함하는 클라스이다. Java Applet에서 주클라스는 체계클라스 Applet의 하위클라스이다. 주클라스는 Java 프로그람이 실행되는 입력점이다. 동일한 Java프로그람에서 정의한 일부 클라스사이에는 엄격한 론리적관계가 있어야 할 필요는 없다. 그러나 그것들은 보통 함께 협동하여 동작하며 매개 클라스들은 다른 클라스에서 정의한 정적속성이나 메쏘드를 사용할수 있다.

Java프로그람에서 클라스정의에는 예약어 class를 사용하며 매개 클라스의 정의는 클라스머리부정의와 클라스본체정의로 구성된다. 클라스본체부분에서는 정적속성과 메쏘드를 정의하며 이것들은 클라스의 성원들이다. 여기서 메쏘드는 다른 고급언어의 함수와 류사하며 정적속성은 변수와 류사하다. 클라스머리부에서는 클라스이름선언외에 클라스의 계승속성을 표현할수 있다.

다른 고급언어와 마찬가지로 명령문은 Java프로그람을 구성하는 기본단위의 하나이다. 매 Java명령문은 구분기호(;)로 결속하며 이 구성은 Java의 문법규칙에 부합되여야 한다. 클라스와 메쏘드에 대한 모든 명령문은 대괄호를 써서 묶어야 한다. 정적속성선언문외에 다른 구체적인 조작을 실행하는 명령문은 클라스메쏘드의 대괄호안에 있어야 하며 메쏘드를 뛰여넘어 클라스에서 고립적으로 직접 작성할수 없다.

제2절. 자료형, 변수와 상수

- Java의 문자형에서는 새로운 국제코드방안 Unicode(16bit)를 리용한다.
- Java에는 C++언어에서와 같은 주소지적자형변수가 없다.

2.2.1. 자료형

표 2-1에 Java에서 정의한 기본자료형을 주었는데 Java의 자료형이 C언어와 비슷하다는것을 알수 있다. 다른 점은 우선 Java의 기본자료형의 크기가 쏘프트하드웨어기반의 종류에 따라 변하지 않는다는것이다. 다음으로 Java의 매 자료형은 기정값을 가지며 따라서 변수는 처음에 해당한 자료형의 기정값을 취한다는것이다. 이와 같은 2가지 점에 Java의 교차기반특성과 안정성이 체현되고있다.

표 2-1. Java의 기본자료형

자료형	예약어	크 기	기정값	값 범 위
론리형	boolean	8	false	true, false
바이트형	byte	8	0	-128~127
문자형	char	16	'\u 000'	'\u 0000'~'\u FFFF'
짧은 옹근수형	short	16	0	-32768~32767
옹근수형	int	32	0	-2147483648~2147483647
긴옹근수형	long	64	0	-9223372036854775808~ 9223372036854775807
류동소수점수형	float	32	0F	1.40129846432481707e-45~ 3.40282346638528860e+38
배정확도형	double	64	0D	4.94065645841246544e-324~ 1.79769313486231570e+308d

boolean은 론리형자료를 표시하는 자료형이며 boolean형의 변수나 상수가 취하는 값은 true와 false이다. 여기서 true는 《참》을 의미하며 false는 《거짓》을 의미한다. byte는 2진자료형으로서 매 byte형의 상수나 변수에는 8bit의 2진정보가 포함된다.

Java의 문자형 char는 다른 언어와 좀 다르다. C언어 등의 문자형은 ASCII코드를 리용하며 매 자료는 8bit의 길이를 차지하고있고 다 합해서 256개의 서로 다른 문자를 표시할수 있다. 실례로 문자 A에 대응하는 ASCII코드는 65이다. ASCII코드는 국제표준의 코드방식이며 콤퓨터, 통신 등 령역에서 광범히 응용되고있다. 그러나 ASCII코드 역시 일정한 제한성을 가지고있는데 가장 전형적인 실례가 한자와 같은

동양문자의 처리이다. 한자는 8bit의 코드를 사용하는것으로는 부족하다. 그러므로 전통적인 처리방법은 2개의 8bit문자자료를 리용하여 하나의 한자를 표시하는것인데 이것은 문자의 표현, 처리, 절환 등 측면에서 불편하다.

이 문제해결을 위하여 Java의 문자형에서는 새로운 국제코드방안－Unicode를 리용하였다. 매 Unicode는 16bit를 가지고있으며 포함된 정보량은 ASCII코드에 비해 두배이다. 동양문자인가, 서양문자인가에 관계없이 통일적으로 한개 문자로 표현할수 있다. Unicode방안을 리용하였기때문에 여러 어종을 처리하는 능력이 아주 높으며 Java프로그람은 서로 다른 언어에 기초한 기반에서 원활한 이식을 실현할수 있으므로 사용이 편리하다.

자료형에서 특별히 강조할것은 우에서 소개한 자료형들이 모두 기본자료형이지만 그에 대응하는 자기의 클라스와 대면 등을 가지고있다는것이다. 실례로 double형에 대응하는 클라스는 Double이며 char형에 대응하는 클라스는 Character이다. 이 클라스들은 기본자료형이 표시하는 일정한 범위, 일정한 격식의 수값을 포함하는 동시에 메쏘드들이 있어 수값에 대한 전문적인 조작도 실현할수 있다.(례하면 문자렬을 배정확도수값으로 절환하는것 등)

2.2.2. 식별부

임의의 변수, 상수, 메쏘드, 객체나 클라스들은 하나의 이름으로 그의 존재를 표식하여야 한다. 이 이름이 바로 식별부(identifier)이다.

식별부는 프로그람작성자에 의하여 자유롭게 지정할수 있으나 일정한 문법규칙에 따를것을 요구한다. Java의 식별부에 대한 정의는 아래와 같은 규칙을 가진다. 식별부는 자모, 수자 그리고 특수문자인 밑선(_)과 $기호들을 조합하여 만들수 있다. 식별부는 반드시 자모, 밑선이나 $기호에 의해 시작하여야 한다. Java는 대소문자를 구별하는 언어이므로 class와 Class, system과 System은 서로 다른 식별부를 의미하며 정의와 사용시 이 점에 특별히 주의하여야 한다.

식별부는 어느정도 그것이 표시하는 변수, 상수, 객체나 클라스의 의미를 반영할수 있도록 지정하여야 한다.

표 2-2에서는 옳은 식별부와 틀린 식별부를 보여주고있다.

표 2-2. 옳은 및 틀린 식별부의 실례

옳은 식별부	틀린 식별부
FirstJavaApplet	1FirstJava Application
MySalary12	Tree&Glasses
_isTrue	-isTrue
$theLastOne	Java Builder
HelloWorld	273.15

2.2.3. 상수

상수는 정해지면 프로그람실행의 전과정에서 변하지 않는다. Java에서 자주 쓰는 상수는 론리상수, 옹근수형상수, 문자상수, 문자렬상수와 류동소수점상수이다.

1) 론리상수

론리상수는 true와 false를 가지는데 각각 참과 거짓을 의미한다.

2) 옹근수형상수

옹근수형상수는 옹근수형변수에 값주기할수 있으며 10진, 8진, 16진수를 리용하여 표시할수 있다. 10진옹근수형상수는 제일 높은 자리수자를 령이 아닌 수값으로 표시하며(례: 100,-50) 8진옹근수형상수는 0으로 시작한 수자로 표시한다.(례: 017은 10진수의 수자 15을 나타낸다.) 또한 16진옹근수형상수는 머리부가 0x로 시작하는 16진값으로 표시한다.(례: 0x2F는 10진수 47을 나타낸다.) 옹근수형상수는 차지하고 있는 기억기의 길이에 따라 일반 옹근수형상수와 긴 옹근수형상수로 구분한다. 여기서 일반 옹근수형상수는 32bit의 길이를 가지며 긴 옹근수형상수는 64bit의 길이를 가진다. 긴 옹근수형상수의 꼬리부분에 대문자 L이나 소문자 l을 표시한다.(례: -386L, 017777l)

3) 류동소수점상수

류동소수점상수는 소수부를 가지는 수값상수이다. 기억기에 차지하고있는 길이에 따라 일반류동소수점상수와 배정확도류동소수점상수로 구분한다. 일반류동소수점상수는 32bit의 기억기를 차지하며 F, f를 리용하여 표시한다.(례: 19.4F, 3.0513E3, 8701.51f) 배정확도류동소수점상수는 64bit의 기억기를 차지하며 D나 d를 달아주거나 수값만으로 표시하기도 한다.(례: 2.433E-5D, 700041.273d, 3.1415) 다른 고급언어와 마찬가지로 류동소수점상수는 일반표시법과 지수표시법의 두가지 표시방법을 가지는데 여기에서는 언급하지 않는다.

4) 문자상수

문자상수는 단인용괄호를 리용한 단일문자를 의미한다. 이 문자는 라틴자모표의 문자일수도 있고 전의부일수도 있으며 표시하려는 문자에 대응하는 8진수나 Unicode일수도 있다. 전의부는 특수한 의미를 가지며 일반방식으로 표현하기 어려운 문자이다.(례: 되돌이, 행바꾸기 등) 이 특수문자들을 표현하기 위하여 Java에서는 특수한 정의를 도입하였다. 모든 전의부는 역사선기호(\)를 써서 시작한다. 뒤에는 한개의 문자를 붙혀 어떤 특정한 전의부로 표시한다. 이것을 표 2-3에 보여주었다.

표 2-3. 전 의 부

인용방법	대응하는 Unicode	의미
'\b'	'\u0008'	backspace
'\t'	'\u0009'	TAB
'\n'	'\u000a'	행바꾸기
'\f'	'\u000c'	형식부
'\r'	'\u000d'	되돌이
'\" '	'\u0022'	쌍인용괄호
'\' '	'\u0027'	단인용괄호
'\\'	'\u005c'	역사선

표 2-3의 두번째 렬에 표시된것은 전의부들에 해당한 Unicode들이다.

문자상수는 8진수로도 표시할수 있는데 실례로 '101'은 8진법으로 한개의 문자상수를 표시하며 Unicode로는 'u0047'이고 'A'에 해당한다.

5) 문자렬상수

문자렬상수는 쌍인용괄호를 써서 표현하는 문자배렬이다. 문자렬에는 전의부를 포함할수 있으며 문자렬의 시작과 끝을 표시하는 쌍인용괄호는 반드시 원천코드의 같은 행에 있어야 한다.

아래에서 몇개의 문자렬상수의 례를 보여준다.

"Hello", "My\nJava", "How are you? 1234", ""

Java에서는 련결조작부 《+》를 사용하여 2개이상의 문자렬상수를 함께 련결할수 있다. 실례로 "How do you do?" + "\n"의 결과는 "How do you do?\n"이다.

2.2.4. 변수

변수는 프로그람의 실행과정에서 값이 변할수 있는 자료이다. 보통 연산의 중간결과나 자료를 보관한다. Java에서 변수는 반드시 선언한 후에 사용하여야 하며 변수의 선언은 변수의 자료형과 변수의 이름을 포함하여야 한다. 필요하다면 변수의 초기값을 지정할수도 있다. 아래의 실례에서는 론리형변수의 선언을 보여준다.

```
boolean m_bFlag = true;
```

변수이름은 m_bFlag이며 초기값은 론리적으로 참이다. 변수의 명령문선언 역시 Java프로그람에서 완전한 명령문이므로 다른 Java명령문과 같이 구분기호(;)로 결속하여야 한다. 아래에 몇개의 변수선언의 실례를 보여준다.

```
char myCharacter = 'B';
long MyLong = -375;
int m_iCount = 65536;
double m_dScore;
```

변수선언을 변수창조라고도 한다. 변수선언문을 실행할 때 체계는 변수의 자료형에 따라 기억기에서 상응한 공간을 내여 변수이름, 초기값 등의 정보를 등록한다. Java에서 변수는 일정한 생존기와 유효범위를 가진다. C언어와 같이 Java는 대괄호를 리용하여 몇개의 명령문을 하나의 명령블로크로 구성하는데 변수의 유효범위는 그것을 선언한 명령문이 있는 명령문블로크이다. 일단 프로그람의 실행이 이 명령문블로크를 벗어나면 변수는 의미를 가지지 않으며 더는 사용할수 없다.

실례 2-1

```
Example 2-1 UseVariable.java
1: public class UseVariable
2: {
3:     public static void main(String args[])
4:     {
5:         boolean b = true;
6:         short si = 128;
7:         int i = -99;
8:         long l = 123456789L;
9:         char ch = 'J';
10:        float f = 3.1415925F;
11:        double d = -1.04E-5;
12:        String s = ("안녕하십니까!");
13:        System.out.println("론리형변수 b = " + b);
14:        System.out.println("짧은 옹근수형변수 si = " + si);
15:        System.out.println("옹근수형변수 i = " + i);
16:        System.out.println("긴 옹근수형변수 l = " + l);
17:        System.out.println("문자형변수 ch = " + ch);
18:        System.out.println("류동소수점형변수 f = " + f);
19:        System.out.println("배정확도형변수 d = " + d);
20:        System.out.println("문자렬변수 s = " + s);
21:    }
22:}
```

프로그람설명

실례 2-1은 문자대면부의 Java Application프로그람이다. 여기에서 몇개의 변수를 정의하고 초기값을 주었다. 10행에서는 류동소수점상수 3.1415925F를 사용하였다. 11행의 배정확도상수는 과학적인 계산법을 사용할 때 표시한다. 12행은 문자렬객체를 정의하고있는데 String은 기본자료형이 아니고 체계가 정의한 클라스이다. 매

문자렬변수는 실제상 문자렬객체이다. 그러나 문자렬은 늘 사용되는 객체이므로 그의 선언과 창조는 12행과 같은 형식으로 간단히 할수 있다. 13-20행에서는 System.out.println( )메쏘드를 리용하여 앞에서 정의한 모든 변수에 대한 값을 출력한다.

그림 2-1은 실례 2-1의 실행결과이다.

```
D:\Javatextbook\Test>javac UseVariable.java

D:\Javatextbook\Test>java UseVariable
론리형변수 b=true
짧은 옹근수형변수 si=128
옹근수형변수 i=-99
긴 옹근수형변수 l=123456789
문자형변수 ch=J
류동소수점형변수 f=3.1415925
배정확도형변수 d=-1.04E-5
문자렬변수 s=안녕하십니까!

D:\Javatextbook\Test>
```

그림 2-1. 실례 2-1의 실행결과

실례 2-2

```
Example 2-2 getNumber.java
1: import java.io.*;
2:
3: public class getNumber
4: {
5:     public static void main(String args[])
6:     {
7:         int i = 0;
8:         String s;
9:
10:        try{
11:            System.out.print("옹근수 한개를 입력하십시오:");
12:            BufferedReader br =
13:                    new BufferedReader(new InputStreamReader(System.in));
14:            s = br.readLine( );
15:            i = Integer.parseInt(s);
16:        }catch(IOException e){ }
17:        System.out.print("수 " + i + "를 입력하였습니다.");
```

```
18:        System.out.println("\t옳습니까?");
19:    }
20:}
```

프로그람설명

실례 2-2는 사용자가 건반입력한 문자렬을 접수한 다음 그것을 옹근수자료로 바꾸어 출력한다. 문자렬 《2004》를 입력하면 실례 2-2의 15행에서 체계가 정의한 메쏘드 Integer.parseInt()를 리용하여 그것을 수자로 바꾼다. 여기서 Integer는 체계가 정의한 클라스이며 기본자료형 int에 대응한다. parseInt( )는 Integer클라스의 메쏘드이다. 수자, 문자로 구성된 문자렬을 옹근수형수자로 절환할수 있다.

11행과 17행에서 리용한 새로운 출력메쏘드 System.out.print( )는 사용방법과 작용이 System.out.println( )과 기본적으로 같으며 유일한 차이는 자료를 출력한 후 되돌이하지 않는다는것이다. 18행은 전의부 '\t'를 사용하여 일정한 간격을 두고 출력하게 한다. 그림 2-2는 실례 2-2의 실행결과이다.

```
D:\Javatextbook\Test>javac getNumber.java

D:\Javatextbook\Test>java getNumber
옹근수 한개를 입력하십시오:2004
수 2004를 입력하였습니다.        옳습니까?

D:\Javatextbook\Test>
```

그림 2-2. 실례 2-2의 실행결과

실례 2-3

```
Example 2-3 getDouble.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class getDouble extends Applet implements ActionListener
6: {
7:     Label prompt;
8:     TextField input;
9:     double d = 0;
10:
11:    public void init( )
```

```
12:     {
13:         prompt = new Label("류동소수점수 한개를 입력하십시오:");
14:         input = new TextField(10);
15:         add(prompt);
16:         add(input);
17:         input.addActionListener(this);
18:     }
19:     public void paint(Graphics g)
20:     {
21:         g.drawString("입력한 자료:" + d, 10, 50);
22:     }
23:     public void actionPerformed(ActionEvent e)
24:     {
25:         d = Double.valueOf(input.getText( )).doubleValue( );
26:         repaint( );
27:     }
28:}
```

그림 2-3은 실례 2-3의 실행결과이다.

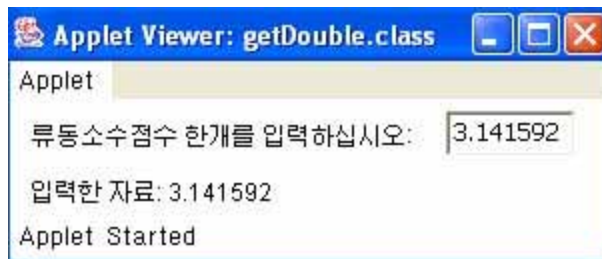



그림 2-3. 실례 2-3의 실행결과

프로그람설명

실례 2-3은 도형사용자대면부의 Java Applet프로그람이다. 사용자가 본문마당객체 input에서 문자렬을 입력하고 되돌이한 후에 프로그람은 사용자가 입력한 문자렬(실례에서는 3.141592)을 접수하고 25행의 Double.valueOf( ).doubleValue( )메쏘드를 리용하여 이 문자렬을 류동소수점자료로 바꾸어 배정확도변수 d에 값주기한다. 26행은 Applet클라스에서 체계가 정의한 repaint( )메쏘드를 사용하고있으며 이 메쏘드는 19-22행의 paint( )메쏘드를 리용하여 변수 d의 수값을 현시한다.

제3절. 표현식

- **강제형변환은 변수를 긴 자료형으로부터 짧은 자료형으로 바꿀 때에만 한다.**
- **Java의 연산조작은 C++언어와 기본적으로 같다.**

표현식은 변수, 상수, 객체, 메쏘드호출과 연산자로 구성된 식이다. 문법규칙에 부합되는 표현식은 번역체계가 리해하고 집행, 계산할수 있으며 표현식의 값은 연산후에 얻는 결과이다. 표현식을 구성하는 Java연산자에는 여러가지가 있는데 여러 종류의 풍부한 연산조작을 나타내고있다. 여기에는 값주기연산, 산수연산, 관계연산, 론리연산 등이 있다.

2.3.1. 값주기와 강제형변환

값주기연산자는 값주기연산에 대응하며 프로그람안의 변수나 객체의 내용에 값주기한다. 간단한 값주기연산은 표현식의 값을 직접 변수나 객체에 값주기한다. 값주기연산자는 《=》이다. 형식은 아래와 같다.

변수 혹은 객체=표현식;

여기서 값주기기호 오른변의 표현식은 상수이거나 다른 변수나 객체 및 메쏘드의 귀환값일수도 있다. 아래에 간단한 값주기연산의 실례를 소개한다.

```
i = 0;
j = 0;
k = i + j + 5;
MyFirstString = MyDouble.toString( );
MySecondString = MyFirstString;
```

주의해야 할것은 값주기기호의 왼변이 객체이름일 때 값주기연산자는 오른변표현식이 얻은 객체의 인용을 그것에 값주기하며 이 객체에 대해서 새로운 기억기공간을 할당하는것이 아니라 오른변객체의 모든 내용을 그것에 값주기한다.

값주기연산자를 사용할 때 값주기기호의 왼변의 자료형과 오른변의 자료형이 일치되지 않는 경우가 있을수 있는데 이때 값주기기호의 오른변의 자료형을 왼변의 자료형으로 바꾸어야 한다. 즉 **강제형변환**(casting)을 진행하여야 한다.

Java에서의 형변환은 변수를 기억기를 적게 차지한 짧은 자료형으로부터 기억기를 비교적 많이 차지하는 긴 자료형으로 변환하는 경우이면 형변환선언을 하지 않아도 된다. 그러나 변수를 긴 자료형으로부터 짧은 자료형으로 바꿀 때에는 반드시 강제형변환을 하여야 한다. 아래의 실례에서는 16bit의 바이트형변수 MyByte와 32bit의 옹근수형변수 MyInteger를 각각 정의하고있다.

```
byte MyByte = 10;
int MyInteger = -1;
```

만일 MyByte의 값을 MyInteger에 대입하는 경우에는 아래와 같이 직접 쓸수 있다.

MyInteger = MyByte;

그러나 MyInteger의 값을 MyByte에 주려면 반드시 다음과 같이 써야 한다.

MyByte = (byte)MyInteger;

우선 변수 MyInteger에 보존한 수값의 자료형을 int로부터 byte로 변환한 다음에야 MyByte에 값주기한다. 여기서 (byte)는 강제형변환이다. 일반적인 형식은 아래와 같다.

(자료형) 변수이름 혹은 표현식

2.3.2. 산수연산

산수연산은 수값형에 대하여 진행하는 연산이다. 산수연산자는 요구되는 연산수에 따라 그의 개수가 같지 않으며 2항연산자와 단항연산자로 구분한다.

1) 2항연산자

여기에서 2가지 주의할 문제가 있다.

· 옹근수형(int, long, short)자료만을 가질 때에야 나머지연산을 진행할수 있으며 float와 double형은 나머지연산을 할수 없다.

· 2개의 옹근수형자료를 나눌 때 결과는 상의 옹근수부이며 소수부는 잘라버린다. 만일 소수부를 남기려면 나누기연산에서 강제형변환을 하여야 한다. 실례로 1/2의 결과는 0이나 ((float)1)/2의 결과는 0.5이다.

표 2-4. 2항산수연산자

연산자	연 산	례	기 능
+	더하기	a + b	a와 b의 합
-	덜기	a - b	a와 b의 차
*	곱하기	a * b	a와 b의 적
/	나누기	a / b	a와 b의 상
%	나머지	a % b	a를 b로 나눌 때 얻어지는 나머지

2) 단항연산자

단항연산자의 연산수는 오직 1개이며 연산자에는 3개의 단항연산자가 있다.

표 2-5. 단항연산자

연산자	연산	례	기능
++	1을 더하기	a++ 또는 ++a	a = a + 1
--	1을 덜기	a-- 또는 --a	a = a - 1
-	반대수 얻기	-a	a = -a

단항연산자에서 자동증가, 자동감소연산자는 연산수의 앞에 놓일수도 있고 뒤에 놓일수도 있다. 단항연산을 진행하는 표현식이 복잡한 표현식의 내부에 위치하고있을 때 단항연산자의 위치는 단항연산과 복잡한 표현식사이의 실행선후차를 결정한다. 아래의 실례에서와 같이 단항연산자가 연산수의 앞에 있으면 단항연산을 먼저 실행하고 변수의 값을 수정한 후에 이 값을 리용하여 복잡한 표현식의 연산에 참가한다.

```
int x = 2 ;
int y = (++x) * 3;
```

연산실행결과는 x=3, y=9이다. 아래의 실례에서는 단항연산자가 연산수의 뒤에 놓여있으므로 복잡한 표현식의 값을 먼저 계산하고 마지막에 변수의 값을 다시 수정한다.

```
int x = 2;
int y = (x++) * 3;
```

연산실행결과는 x=3, y=6이다. 보는바와 같이 이 연산자의 위치가 다를 때 연산수변수에는 영향이 없어도 전체 표현식의 값은 총적으로 변경된다는것을 알수 있다.

실례 2-4

```
Example 2-4 TestArithmetic.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class UseArithmetic extends Applet implements ActionListener
6: {
7:      Label prompt;
8:      TextField input1, input2;
9:      Button btn;
10:     int a = 0, b = 1;
11:
12:     public void init( )
13:     {
14:         prompt = new Label("두개의 옹근수형자료를 입력하십시오:");
15:         input1 = new TextField(5);
16:         input2 = new TextField(5);
17:         btn = new Button("계산");
18:         add(prompt);
19:         add(input1);
20:         add(input2);
21:         add(btn);
```

```
22:        btn.addActionListener(this);
23:    }
24:    public void paint(Graphics g)
25:    {
26:        g.drawString(a + "+" + b + "=" + (a + b), 10, 50);
27:        g.drawString(a + "-" + b + "=" + (a - b), 10, 70);
28:        g.drawString(a + "*" + b + "=" + (a * b), 10, 90);
29:        g.drawString(a + "/" + b + "=" + (a / b), 10, 110);
30:        g.drawString(a + "%" + b + "=" + (a % b), 10, 130);
31:    }
32:    public void actionPerformed(ActionEvent e)
33:    {
34:        a = Integer.parseInt(input1.getText( ));
35:        b = Integer.parseInt(input2.getText( ));
36:        repaint( );
37:    }
38:}
```

프로그람설명

실레 2-4는 도형사용자대면부에서의 Java Applet프로그람이며 이것은 두개의 본문마당객체 input1과 input2를 리용하여 사용자가 입력한 두개의 자료를 접수한다.

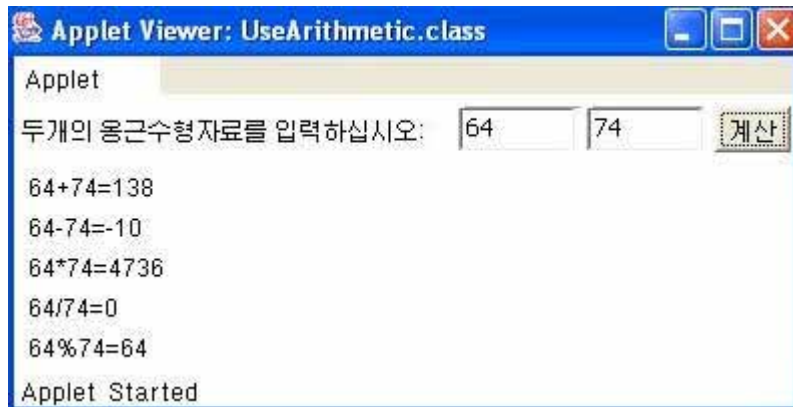



그림 2-4. 실례 2-4의 실행결과

사용자가 《계산》단추를 찰칵할 때 프로그람은 두개의 문자렬을 옹근수형자료로 바꾸어 같은 클라스안의 두개 변수 a와 b에 값주기하며 다음에 repaint( )메쏘드를 통해서 24-31행의 paint메쏘드를 사용하여 연산수 a와 b의 사칙연산결과를 출력한다. 그림 2-4는 실례 2-4의 실행결과이다.

2.3.3. 관계연산

관계연산은 2개 자료사이의 크기관계를 비교하는 연산으로서 보통 리용되는 관계연산자들을 표 2-6에 주었다.

표 2-6. 관계연산자

연산자	연 산
==	같다.
!=	같지 않다.
>	크다.
<	작다.
>=	크거나 같다.
<=	작거나 같다.

관계연산의 결과는 론리형으로서 참 또는 거짓이다. 실례로

```
int x=5; y=7;
boolean b=(x == y);
```

이때 b의 초기값은 false이다. 여기서 같기기호와 값주기기호를 정확히 구분하여야 한다.

실례 2-5

```
Example 2-5 UseRelation.java
24:    public void paint(Graphics g)
25:    {
26:        g.drawString(a + ">" + b + "=" + (a > b), 10, 50);
27:        g.drawString(a + "<" + b + "=" + (a < b), 10, 70);
28:        g.drawString(a + ">=" + b + "=" + (a >= b), 10, 90);
29:        g.drawString(a + "<=" + b + "=" + (a <= b), 10, 110);
30:        g.drawString(a + "==" + b + "=" + (a == b), 10, 130);
31:        g.drawString(a + "!=" + b + "=" + (a != b), 10, 150);
32:    }
```

그림 2-5는 실례 2-5의 실행결과이다. 실례 2-5는 실례 2-4에 기초하여 24-31행의 paint( )메쏘드만을 수정하였으며 사용자가 입력한 두개의 옹근수사이의 관계를 비교하고 비교결과를 출력한다.

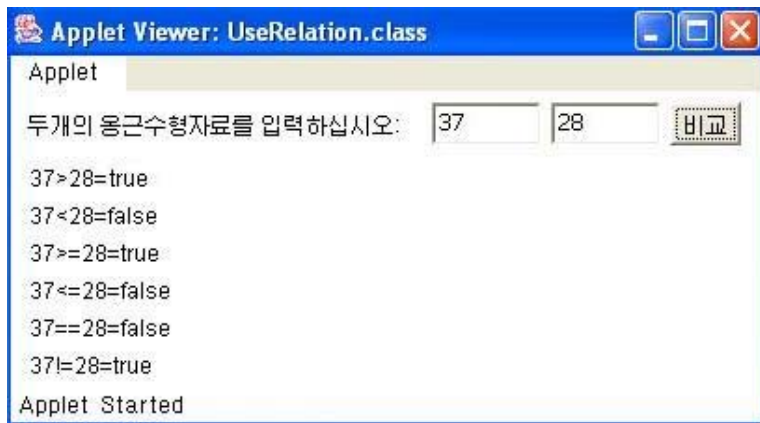



그림 2-5. 실례 2-5의 실행결과

2.3.4. 론리연산

론리연산은 론리형자료에 대하여 AND, OR연산을 진행하며 연산결과는 론리형이다. 보통 리용하는 론리연산자들을 표 2-7에 주었다.

표 2-7. 론리연산자

연산자	연 산	례	연산규칙
&&	론리적	x && y	x, y가 모두 참일 때 참
||	론리합	x || y	x, y가 모두 거짓일 때 거짓
!	론리부정	!x	X가 참이면 거짓, 거짓이면 참

```
int x = 3, y = 5;
boolean b = x > y && x++ == y--;
```

론리형변수 b의 값을 계산할 때 우선 론리적(&&)의 왼쪽관계표현식 x > y를 계산하면 결과가 거짓이다. 론리적연산규칙에 따라 AND연산에 참가하는 두 표현식의 값이 모두 참일 때에만 마지막 결과가 참으로 된다. 그러므로 연산자 &&의 오른쪽표현식결과가 어떻든지간에 전체식의 값은 거짓으로 되며 오른변의 표현식은 계산집행하지 못하게 된다. 최종적으로 3개변수의 값은 각각 x는 3, y는 5, b는 false이다.

마찬가지로 론리합(||)에 대하여 왼쪽표현식의 연산결과가 참이면 전체의 연산결과는 참으로 되며 오른변의 표현식은 계산하지 않아도 된다.

실례 2-6

```
Example 2-6 UseLogical.java
…
10: boolean a=true, b=false;
…
24: public void paint(Graphics g)
25: {
26:         g.drawString(a + "&&" + b + "=" + (a && b), 10, 70);
27:         g.drawString(a + "||" + b + "=" + (a || b), 10, 110);
28:         g.drawString("!" + b + "=" + (!b), 10, 110);
29: }
30: public void actionPerformed(ActionEvent e)
31: {
32:         a=Boolean.valueOf(input1.getText( )).booleanValue( );
33:         b=Boolean.valueOf(input2.getText( )).booleanValue( );
34:         repaint( );
35: }
```

프로그람설명

실례 2-6은 실례 2-4에 기초하여 수정한것이다. 30-35행의 사건처리메쏘드를 수정하여 체계가 정의한 메쏘드 Boolean.valueOf( ).booleanValue( )를 리용하면 사용자가 입력한 문자렬을 론리형자료로 바꾼다. 24-29행의 paint( )메쏘드를 수정하여 2개의 론리연산결과를 현시한다. 그림 2-6은 실례 2-6의 실행결과이다.

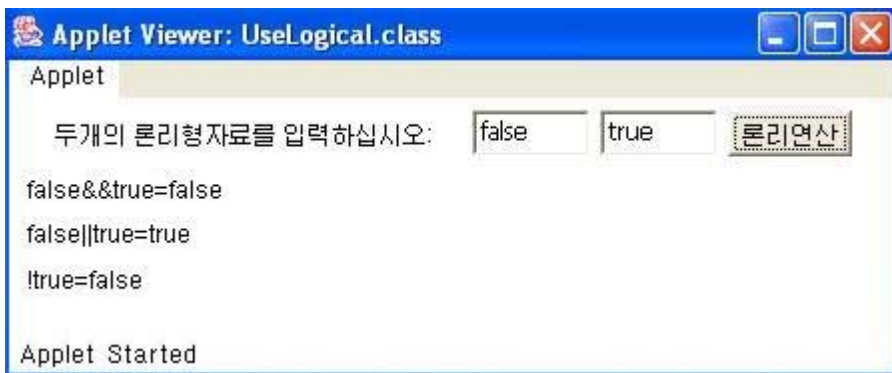



그림 2-6. 실례 2-6의 실행결과

2.3.5. 비트연산

비트연산은 연산수에 대하여 2진비트를 단위로 하여 진행하는 연산이며 비트연산의 연산수와 결과는 옹근수형이다. 일반적인 프로그람작성에서는 많이 리용되지 않지만 화상처리나 조종체계를 만드는데 자주 리용된다.

1) 비트론리연산

비트론리연산자는 비트별로 AND, OR연산을 진행한다. 비트론리연산자들을 표 2-8에 보여주었다.

표 2-8. 비트론리연산자

연산자	연 산	례	연산규칙
&	비트론리적	x & y	x와 y의 대응하는 비트가 모두 참일 때 참
|	비트론리합	x | y	x와 y의 대응하는 비트가 모두 거짓일 때 거짓
ˆ	배타적론리합	x ˆ y	대응하는 비트값이 서로 다를 때 참
~	비트반전	~x	x를 비트별로 반전한다.

```
byte b1 = 0x11;          // b1은     00010001
byte b2 = 0x33;          // b2은     00110011
int result = b1 & b2     // 결과값은     00010001(17)
int result = b1 | b2     // 결과값은 00110011(51)
int result = b1 ^ b2     // 결과값은 00100010(34)
int result = ~b2         // 결과값은     11001100(-19)
```

2) 비트옮김연산

비트옮김연산은 어떤 변수가 포함하는 매 비트를 지정한 방향에 따라 지정한 비트수만큼 이동하는것이다. 표 2-9에서 3개의 비트옮김연산자를 보여주었다.

표 2-9. 비트옮김연산자

연산자	연 산	례	연산규칙
>>	오른쪽옮김	x >> a	x의 매 비트를 abit 오른쪽으로 옮긴다.
<<	왼쪽옮김	x << a	x의 매 비트를 abit 왼쪽으로 옮긴다.
>>>	부호 붙지 않은 오른쪽옮김	X >>> a	x의 매 비트를 abit 오른쪽으로 옮기고 왼변의 빈 비트는 일률적으로 령을 채운다.

비트옮김연산의 실례를 표 2-10에 보여준다.

표 2-10. 비트옮김연산의 실례

x(10진표시)	2진표시	x << 2	x >> 2	x >>> 2
30	00011110	01111000	00000111	00000111
-17	11101111	10111100	11111011	00111011

우의 표에서 볼수 있는것처럼 부호붙은 오른쪽옮김에서는 오른쪽으로 옮긴 다음에 왼쪽의 남은 빈 비트에 원래의 부호가 채워진다. 즉 정수는 0이, 부수는 1이 채워진다. 부호가 없는 오른쪽옮김에서는 오른쪽옮김후 왼쪽의 빈 비트에 일률적으로 0이 채워진다.

2.3.6. 기타 연산자

1) 3항조건연산자

Java에서 3항조건연산자(?)는 c언어에서와 완전히 같으며 다음의 형식으로 사용한다.

x ? y : z

우선 표현식 x의 값을 계산하고 만일 x가 참이면 전체 3항연산의 결과가 표현식 y의 값으로 된다. 만일 x가 거짓이면 전체 연산결과는 표현식 z의 값으로 된다. 아래에 실례를 주었다.

```
int x = 5, y = 8, z = 2;
int k = x < 3 ? y : z;   //k는 z의 값을 취하여 결과는 2이다.
int y = x > 0 ? x : -x;   //y는 x의 절대값이다.
```

2) 복잡한 값주기연산자

복잡한 값주기연산자는 우선 어떤 연산을 진행한 후에 다시 연산의 결과를 값주기한다. 표 2-11은 복잡한 값주기연산자들을 모두 보여준다.

표 2-11. 값주기연산자

연산자	례	기능
+=	x += a	x = x + a
—=	x -= a	x = x - a
*=	x *= a	x = x * a
/=	x /= a	x = x / a

연산자	례	기능
%=	x %= a	x = x % a
&=	x &= a	x = x & a
|=	x |= a	x = x | a
^=	x ^= a	x = x ^ a
<<=	x <<= a	x = x << a
>>=	x >>= a	x = x >> a
<<<=	x <<<= a	x = x <<< a

3) 객체연산자

객체연산자 instanceof는 객체가 클라스나 하위클라스를 지정하는 어떤 실례에 속하는가 속하지 않는가를 판정하는데 쓰이며 만일 옳으면 true를 귀환하고 그렇지 않으면 false를 귀환한다.

```
boolean b = MyObject instanceof TextField;
```

2.3.7. 연산자의 우선권과 결합성

연산자의 우선권은 표현식에서 서로 다른 연산이 실행될 때의 선후차결정을 말한다. 관계연산자의 우선권이 론리연산자보다 높으며 x>y && !z는 (x>y)&&(!z)와 우선권이 같다.

연산자의 결합성은 병렬적인 서로 같은 연산의 선후집행순서를 결정한다. 실례로 왼쪽결합의 《+》에 대하여서는 x + y + z 와 (x + y) + z는 같으며 오른쪽결합의 《!》에 대하여 !!x는 !(!x)와 같다.

표 2-12는 Java에서 주요연산자의 우선권과 결합성을 보여준다.

표 2-12. Java연산자의 우선권과 결합성

우선권	표 현	연산자	결합성
1	최고우선권	. { } ()	왼쪽/오른쪽
2	단항연산	— ~ ! ++ --	오른쪽
3	곱하기나누기산수연산	* / %	왼쪽
4	더하기덜기산수연산	+ —	왼쪽
5	비트옮김연산	>> << >>>	왼쪽
6	크기관계연산	< <= > >=	왼쪽
7	같기관계연산	== !=	왼쪽

8	비트론리적	&	왼쪽
9	배타적론리합	ˆ	왼쪽
10	비트론리합	|	왼쪽
11	론리적	&&	왼쪽
12	론리합	||	왼쪽
13	3항조건연산	? :	오른쪽
14	값주기연산	=연산자=	오른쪽

2.3.8. 주석

주석은 프로그람에서 없어서는 안될 부분이다. Java의 주석에는 2가지가 있다.

하나는 행주석 《//》인데 《//》의 머리부부터 시작하여 이 행마지막까지의 모든 문자는 체계에서 주석으로 리해하고 번역하지 않는다. 실례로

//This a testprogram of what is to be done

다른 하나의 주석은 블로크주석 《/*》와 《*/》인데 여기서 《/*》은 블로크주석의 시작을 의미하며 《*/》은 블로크주석의 끝을 의미한다. 실례로

/* 프로그람이름:

항목이름:

작성시간:

기능:

입력/ 출력: */

제4절. 흐름조종문

- 분기명령문(if…else…, switch…case)
- 순환명령문(while, do…while, for)
- 뛰여넘기명령문(continue, break, return)

흐름조종명령문은 프로그람에서 매 명령문의 집행순서를 조종하는 명령문으로서 아주 기본적이고 관건적인 부분이다. 가장 중요한 프로그람흐름조종방식에는 3가지 흐름구조가 있다.

2.4.1. 구조화프로그람설계의 기본흐름

구조화프로그람설계의 가장 기본적인 원칙은 다음과 같다. 임의의 프로그람은 3가지 기본흐름구조로 구성된다. 즉 순서구조, 분기구조, 순환구조로 구성된다. 이 3가지 구조의 구성을 그림 2-7에 보여주었다.

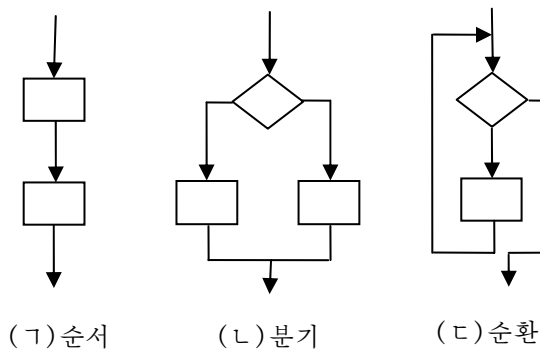



그림 2-7. 구조화프로그람설계의 3가지 기본구조

순서구조는 3가지 구조에서 가장 간단한 류형으로서 명령문은 씌여진 순서에 따라 집행되며 **분기구조**는 선택구조라고도 하는데 이것은 계산에서 얻은 표현식의 값에 따라 어느 흐름을 선택집행해야 하는가 하는 분기를 판단한다. **순환구조**는 어떤 조건하에서 일부분의 명령문들을 반복집행하는 흐름구조이다. 이 3가지 구조는 프로그람에 대한 국부블로크의 기본골격을 형성한다.

Java언어는 객체지향언어이지만 국부적인 명령문블로크내부에서는 여전히 구조화프로그람설계의 기본흐름구조를 써서 명령문을 구성하며 상응한 론리적기능을 완성한다. Java의 명령블로크는 하나의 대괄호로 묶은 몇개의 명령문들의 모임이다. Java에는 분기구조를 전문적으로 실현하는 분기명령문과 순환구조를 실현하는 순환명령문이 있다.

2.4.2. 분기명령문

분기명령문에는 2가지가 있는데 하나는 쌍분기를 실현하는 if명령문이고 다른 하나는 다중분기를 실현하는 switch명령문이다.

1) if명령문

if명령문의 일반형식은 아래와 같다.

```
if (조건표현식)
    명령문블로크; //if분기
else
    명령문블로크; //else분기
```

여기서 조건표현식은 프로그람의 흐름방향을 선택판단하며 만일 조건표현식의 값이 참이면 if분기의 명령문블로크를 집행하고 그렇지 않으면 else분기의 명령문블로크를 집행한다. 프로그람을 작성할 때 else분기를 쓰지 않을수도 있는데 이때 조건표현식의 값이 거짓이면 if분기를 뛰여넘어 if명령문뒤의 다른 명령문을 직접 집행한다. 문법형식은 아래와 같다.

```
if (조건표현식)
    명령문블로크; //if분기
기타 명령문;
```

실례 2-7

```
Example 2-7 FindMax.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class FindMax extends Applet implements ActionListener
6: {
7:     Label result;
8:     TextField in1, in2, in3;
9:     Button btn;
10:    int a = 0, b = 0, c = 0, max;
11:
12:    public void init( )
13:    {
14:        result = new Label("비교하려는 3개의 옹근수를 먼저 입력하십시오");
15:        in1 = new TextField(5);
```

```
16:         in2 = new TextField(5);
17:         in3 = new TextField(5);
18:         btn = new Button("비 교");
19:         add(in1);
20:         add(in2);
21:         add(in3);
22:         add(btn);
23:         add(result);
24:         btn.addActionListener(this);
25:     }
26:     public void actionPerformed(ActionEvent e)
27:     {
28:         a = Integer.parseInt(in1.getText( ));
29:         b = Integer.parseInt(in2.getText( ));
30:         c = Integer.parseInt(in3.getText( ));
31:         if(a > b)
32:             if(a > c)
33:                 max = a;
34:             else
35:                 max = c;
36:         else
37:             if(b > c)
38:                 max = b;
39:             else
40:                 max = c;
41:         result.setText("최 대 값:" + max);
42:     }
43:}
```

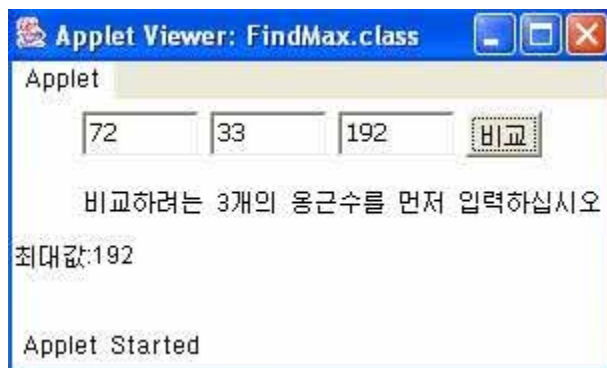



그림 2-8. 실례 2-7의 실행결과

프로그람설명

실례 2-7의 프로그람은 사용자가 입력한 3개 옹근수를 접수하고 사용자가 《비교》단추를 찰칵하면 actionPerformed()메쏘드를 호출하여 여기서 수들을 비교하고 최대값을 출력한다. 31-40행에서는 if명령문을 사용하여 3개 옹근수를 비교해서 최대값을 얻는다. 41행에서는 Label클라스객체 result에서 체계가 정의한 setText메쏘드를 호출하여 최대값을 출력한다.(그림 2-8)

2) switch명령문

switch명령문은 다중분기명령문이다. 일반형식은 아래와 같다.

```
switch(표현식)
{
    case 판단값 1: 명령문블로크 1    //분기1
    case 판단값 2: 명령문블로크 2    //분기2
    … …
    case 판단값 n: 명령문블로크 n   //분기n
    default:  명령문블로크 n+1      //분기 n+1
}
```

switch명령문이 집행될 때 우선 표현식의 값을 계산하는데 이 값은 반드시 옹근수형이나 문자형이여야 하며 동시에 매 case분기의 판단값형과 일치하여야 한다. 표현식의 값을 계산한 후에는 그것을 첫번째 case분기의 판단값과 비교하고 같으면 프로그람이 첫번째 case분기의 명령문블로크에 들어간다. 그렇지 않으면 다시 표현식의 값을 두번째 case분기와 비교하며 이러한 과정을 반복해나간다. 만일 표현식의 값이 임의의 어떤 case분기와도 일치하지 않으면 마지막의 default분기로 가서 집행한다. default분기가 없는 경우에는 switch명령문에서 탈퇴한다.

주의해야 할것은 switch명령문의 매 case판단은 흐름분기의 입력점을 담당하고있을뿐이지 분기를 지정하는 출력점은 담당하지 않는다는것이다. 분기의 출력점은 프로그람작성자가 상응한 뛰여넘기명령문으로 작성하여야 한다. 아래에 실례를 주었다.

```
switch(MyGrade)
{   case      'A'  :    MyScore = 5;
    case      'B'  :    MyScore = 4;
    case      'C'  :    MyScore = 3;
    default        :    MyScore = 0;}
```

변수 MyGrade의 값이 A라고 가정하면 switch문을 실행한 후에 변수 Myscore의 값이 무엇으로 되겠는가? 5가 아니고 0인데 무엇때문인가? case판단은 분기의 입력점을 담당할뿐이므로 표현식의 값은 첫째 case분기판단값과 비교한 후에 흐름이 첫번째 분기에 진입하여 Myscore의 값을 5로 놓는다. 전문적인 분기출력을 가지지 않으므로 프로그람흐름은 계속 아래의 분기를 축차적으로 실행하며 Myscore값은 순차적으로 4,

3으로 되고 마지막에는 0으로 된다. 만일 프로그람의 론리구조가 분기의 선택을 정상적으로 완성하도록 하자면 매 분기에 대하여 탈퇴문을 써놓아야 한다. 수정하면 아래와 같다.

```
switch(MyGrade)
{   case    'A'  :   MyScore = 5; break;
    case    'B'  :   MyScore = 4; break;
    case    'C'  :   MyScore = 3; break;
    default      :   MyScore = 0; }
```

break는 뛰여넘기명령문으로서 그의 구체적인 사용방법은 아래에서 서술한다.

break명령문을 리용하여 매 분기출력을 정의하며 다중분기스위치명령문의 구조를 완성한다.

2.4.3. 순환명령문

순환명령문은 어떤 조건하에서 프로그람을 반복집행하는 흐름구조이며 반복집행되는 프로그람부분을 순환본체부분이라고 부른다. 순환구조는 프로그람에서 아주 중요하고 기본적인 구조이다. 순환명령문에는 3가지 즉 while명령문, do-while명령문, for명령문이 있다. 그것들의 구조를 그림 2-9에 보여주었다.

1) while명령문

while명령문의 일반형식은 아래와 같다.

while (조건표현식)

순환본체부분

여기서 조건표현식의 귀환값은 론리형이며 순환본체부분은 단일명령문일수도 있고 복합명령문일수도 있다.

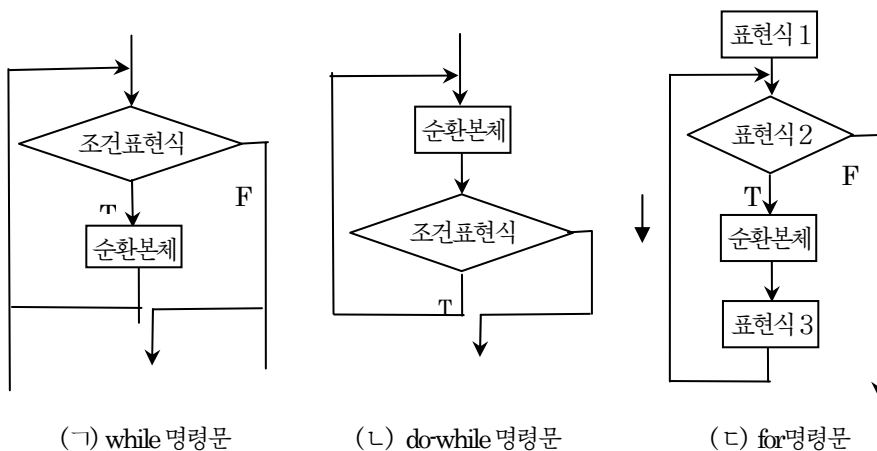



그림 2-9. Java의 3가지 순환명령문

while명령문의 실행과정은 우선 조건표현식의 값을 판단하고 만일 참이면 순환본체부분을 실행하며 순환본체부분실행후에는 다시 조건표현식에로 무조건 이행하여 계산판단한다. 조건표현식의 값이 거짓으로 되는 경우 순환본체부분을 뛰여넘어 while명령문뒤의 명령문을 집행한다.

실례 2-8

```
Example 2-8 Narcissus.java
1: public class Narcissus
2: {
3:      public static void main(String args[])
4:      {
5:           int i, j, k, n = 100, m = 1;
6:
7:           while(n < 1000)
8:           {
9:                i = n / 100;
10:               j = (n – i * 100) / 10;
11:               k = n % 10;
12:               if((Math.pow(i, 3) + Math.pow(j, 3) + Math.pow(k, 3)) == n)
13:                    System.out.println(m++ + "번째 수선화수를 찾음:" + n);
14:               n++;
15:          }
16:     }
17:}
```

프로그람설명

실례 2-8은 문자대면부의 Java Application프로그람이다. 그의 기능은 **수선화수**(Narcissus)를 찾아 출력한다. 수선화수는 3자리수로서 그것의 매 자리수자의 3제곱합은 이 3자리수자체와 같다. 실례로 $371=3^3+7^3+1^3$이면 371은 바로 수선화수이다. 7-15행은 while순환을 정의하고있는데 매 순환은 100부터 999까지의 3자리수 n을 검사하고 n의 1의 자리수, 10의 자리수, 100의 자리수를 각각 옹근수형변수 k, j, i에 넣는다. 12행은 체계가 정의한 메쏘드 Math.pow( )를 리용하여 i, j, k의 3제곱을 계산하여 서로 합한다. 만일 얻은 합이 n과 같으면 수선화수를 출력한다. 계속하여 n에 1을 더하여 다음의 3자리수를 검사한다. 순환이 끝나면 모든 수선화수가 얻어진다. 그림 2-10은 실례 2-8의 실행결과이다.

```
D:\Javatextbook\Test>javac Narcissus.java

D:\Javatextbook\Test>java Narcissus
1번째 수선화수를 찾음:153
2번째 수선화수를 찾음:370
3번째 수선화수를 찾음:371
4번째 수선화수를 찾음:407

D:\Javatextbook\Test>
```

그림 2-10. 실례 2-8 실행결과

2) do-while명령문

do-while명령문의 일반형식은 아래와 같다.

```
do
    순환본체부분
while(조건표현식);
```

do-while명령문의 사용은 while명령문과 거의 류사하며 다른점은 while과 같이 조건표현식의 값을 먼저 판단하는것이 아니라 무조건적으로 순환본체부분을 집행한 다음에 조건표현식을 판단한다는것이다. 표현식의 값이 참이면 순환본체부분을 다시 실행하며 그렇지 않으면 do-while순환에서 탈퇴하여 아래의 명령문을 집행한다.

do-while명령문의 특징은 순환본체부분이 적어도 한번은 집행된다는것이다. 주의해야 할것은 while다음에 반두점(;)을 붙여야 한다는것이다. 아래에 do-while명령문을 사용하는 실례를 보여준다.

실례 2-9

```
Example 2-9 showCharValue.java
1: import java.io.*;
2:
3: public class showCharValue
4: {
5:     public static void main(String args[])
6:     {
7:         char ch;
8:
9:         try
```

```
10:         {
11:             System.out.println("한개 문자를 입력하고 '#'로 끝내십시오.");
12:             do
13:             {
14:                 ch = (char)System.in.read( );
15:                 System.out.println("문자" + ch + "의 값:" + (int)ch);
16:                 System.in.skip(2); //2개의 문자 뛰여넘기
17:             } while(ch != '#');
18:         }
19:         catch(IOException e)
20:         {
21:             System.err.println(e.toString( ));
22:         }
23:     }
24:}
```

프로그람설명

실례 2-9는 문자대면부의 Java Application프로그람으로서 사용자가 입력한 한개 문자를 접수한 후 이 문자의 옹근수형값을 출력한다. 12-17행은 do-while순환이다. 14행은 건반으로부터 한개의 옹근수형자료를 읽어들이고 강제형변환에 의해 문자형으로 변환시켜 문자변수 ch에 값주기한다. 15행은 ch와 그의 옹근수형값을 출력한다. 16행은 사용자가 몇개의 문자를 입력하였을 때 두개의 문자를 뛰여넘어 출력하도록 하는 코드이다. 17행은 사용자가 입력한 문자가 《#》인가를 검사하고 맞으면 순환을 끝내고 그렇지 않으면 14행으로 되돌아와 계속 사용자의 입력을 접수한다. 그림 2-11은 실례 2-9의 실행결과이다.

```
D:\Javatextbook\Test>java showCharValue
한개 문자를 입력하고 '#'로 끝내십시오.
Y
문자Y의 값:89
J
문자J의 값:74
a
문자a의 값:97
A
문자A의 값:65
#
문자#의 값:35

D:\Javatextbook\Test>
```

그림 2-11. 실례 2-9의 실행결과

3) for명령문

for명령문은 Java언어의 순환명령문중에서 기능이 비교적 강하고 널리 사용되는 명령문이다. for명령문의 일반형식은 아래와 같다.

for (표현식 1; 표현식 2; 표현식 3)
    순환본체부분

여기서 표현식 2는 론리형값을 귀환시키는 조건표현식으로서 순환이 계속되는가를 판단하는데 쓰인다. 표현식 1은 순환변수의 초기화와 다른 변수의 작업을 완성한다. 표현식 3은 순환변수를 수정하는데 쓰이며 순환조건을 고친다. 3개의 표현식은 구분기호(;)를 사용하여 갈라놓는다.

for명령문의 집행과정은 다음과 같다. 우선 표현식 1을 계산하여 필요한 초기화 작업을 완성한다. 표현식 2의 값을 판단하여 만일 참이면 순환본체부분을 집행한 후 표현식 3에 다시 돌아가 순환조건을 계산수정한다. 이렇게 한번의 순환이 결속된다.

두번째순환은 표현식 2의 계산판단으로부터 시작하여 표현식값이 여전히 참이면 순환을 계속하며 그렇지 않으면 for명령문에서 탈퇴하여 아래의 명령문을 집행한다. for명령문의 3개 표현식들이 없을수도 있지만 표현식 2가 없으면 무한순환으로 되며 이때는 순환본체부분에 다른 뛰여넘기명령문을 써서 순환을 중지하여야 한다.

실례 2-10

```
Example 2-10 PerfectNum.java
1: public class PerfectNum
2: {
3:      public static void main(String args[])
4:      {
5:          int count = 1;
6:          for(int i = 1; i < 10000; i++)
7:          {
8:              int y = 0;
9:
10:             for(int j = 1; j < i; j++)
11:                 if(i % j == 0)
12:                     y += j;
13:             if(y == i)
14:             {
15:                 System.out.print(i + String.valueOf('\t'));
16:                 count++;
17:                 if(count%3 == 0)
```

```
18:                    System.out.println();
19:            }
20:        }
21:    }
22:}
```

프로그람설명

실례 2-10은 문자대면부의 Java Application프로그람으로서 그의 기능은 10000이내의 모든 완전수를 출력하는것이다. 완전수는 모든 약수들의 합(1을 포함하나 그 자체수를 포함하지 않음)과 같은 수를 말한다. 실례로 6=1*2*3, 6=1+2+3이므로 6은 완전수이다. 6-20행은 for순환을 정의하고 1부터 9999까지의 모든 옹근수가 완전수인가 아닌가를 검사한다. 10-12행은 대순환에 들어있는 소순환이며 수 i의 약수합을 구하는데 쓰인다. 만일 i가 약수들의 합과 같다면 i는 완전수이며 15행에서 그수를 출력한다. 변수 count는 구해진 완전수의 개수를 계산하는데 쓰인다. 17-18행은 매번 2개의 완전수를 출력한 후에 행바꾸기를 한다.(그림 2-12)

```
D:\Javatextbook\Test>java PerfectNum
6        28
496      8128
D:\Javatextbook\Test>
```

그림 2-12. 실례 2-10의 실행결과

2.4.4.뛰여넘기명령문

뛰여넘기명령문은 프로그람집행과정에 흐름의 이행을 실현하는데 쓰인다. 앞의 switch명령문에서 사용하였던 break명령문은 뛰여넘기명령문이다. Java언어는 무조건이행명령문인 goto명령문을 지원하지 않는다. 뛰여넘기명령문에는 3가지 즉 continue명령문, break명령문, return명령문이 있다.

1) continue명령문

continue명령문은 반드시 순환구조안에서 리용되여야 하며 두가지 사용형식이 있다.

하나는 표식이 붙지 않는 continue명령문으로서 현재의 순환을 중지하고 남은 명령문에서 탈퇴하여 순환의 다음 고리에 직접 들어간다. while이나 do-while순환에서 표식이 붙지 않은 continue명령문은 흐름이 조건표현식까지 직접 뛰여넘게 할수 있다. for순환에서 표식이 붙지 않은 continue명령문은 표현식 3에로 뛰여넘기할수 있으며 순환변수를 계산하고 수정한 후에 순환조건을 판단한다.

다른 하나는 표식이 붙은 continue명령으로서 그의 형식은 아래와 같다.

```
continue 표식이름;
```

이 표식이름은 프로그람에서 바깥층순환명령문의 앞에서 정의해야 하며 이 순환구조를 표식하는데 쓰인다. 표식이름은 Java식별부의 규정에 맞아야 한다. 표식이 붙은 continue명령문은 프로그람의 흐름이 표식이 지적하는 순환층으로 들어가게 한다.

아래의 1~100사이의 씨수를 찾는 례제에서 표식이 붙은 continue명령문을 사용하고있다. 만일 옹근수 i의 한개 약수 j를 찾았다면 이 i는 씨수가 아니라는것을 의미한다. 프로그람은 이 순환의 나머지명령문을 뛰여넘어 직접 다음의 순환에 들어가며 수가 씨수인가 아닌가를 검사한다.

```
First_Loop:
for(int i = 1; i < 100; i++)
{
    for(int j = 2; j < i; j++)
    {
        if(i % j == 0)
            continue First_Loop;
    }
    System.out.println(i); //화면표준출력
}
```

2) break명령문

break명령문의 작용은 프로그람의 흐름이 한 명령문블로크내부로부터 뛰여나오게 하거나(례: switch명령문의 분기에서 뛰여나오기) 순환본체내부에서 뛰여나오게 하는것이다. break명령문은 표식이 붙은것과 붙지 않은 명령문으로 나눈다. 표식이 붙은 break명령문의 일반형식은 다음과 같다.

break 표식이름;

이 표식은 어떤 명령문블로크를 표시한다. break명령문을 집행하면 이 명령문블로크다음의 명령문을 집행한다.

표식이 붙지 않은 break명령문은 그것이 있는 switch분기나 제일 안층의 순환본체부분에서 뛰여나와 분기나 순환본체부분뒤의 명령문을 집행한다.

3) return명령문

return명령문의 일반형식은 다음과 같다.

return 표현식;

return명령문은 프로그람흐름을 메쏘드호출로부터 귀환시키는데 있으며 표현식의 값은 호출메쏘드의 귀환값이다. 만일 호출메쏘드가 귀환값을 가지지 않으면 return명령문의 표현식은 생략할수 있다.

제5절. 배렬과 벡토르

- 배렬은 동일한 자료형의 원소들을 일정한 순서로 배렬하여 묶어놓은것이다.
- 벡토르는 서로 다른 형의 원소들이 공존하는 가변길이의 배렬을 리용한다.
- 배렬과 벡토르의 원소들은 간단한 자료형의 변수일수도 있고 클라스의 객체일수도 있다.
- 벡토르는 반드시 창조한 후에 사용한다.

2.5.1. 배렬

배렬은 같은 종류의 자료가 하나의 이름으로 표시되는 자료의 묶음이다. Java에서 배렬의 원소는 간단한 자료형의 변수일수도 있고 어떤 클라스의 객체일수도 있다. 배렬의 주요특징은 아래와 같다.

· 배렬은 동일한 자료형의 원소들의 모임이다.

· 배렬에서의 매 원소는 선후순서를 가진다.

· 매개 배렬원소는 배렬의 이름과 그것이 배렬에서 차지한 위치로 표현된다. 실례로 a[0]은 배렬 a의 첫번째 원소를 의미하며 a[1]은 배렬 a의 두번째 원소를 의미한다.

Java프로그람에서 배렬을 정의하는 조작은 다른 언어와 비교하여보면 일정한 차이를 가진다. 일반적으로 한개의 Java배렬을 창조하는데 아래의 3단계가 요구된다.

1) 배렬선언

배렬선언은 기본적으로 배렬의 이름과 배렬이 포함하고있는 원소의 형이름을 선언한다. 배렬을 선언하는 문법형식에는 2가지가 있다.

배렬원소형　배렬이름[ ];

배렬원소형[ ]　배렬이름;

꺽쇠괄호([ ])는 배렬표식으로서 배렬이름의 뒤에 놓일수도 있고 배렬원소형이름의 뒤에 놓일수도 있다. 이 2개의 정의방법은 아무런 차이가 없다.

```
int MyIntArray[ ];
D200_Card[ ] ArrayOf200Card;
```

2) 배렬공간창조

배렬선언에서는 단지 배렬의 이름과 배렬원소의 형만을 지정하므로 배렬을 실제적으로 사용하려면 그에 대해 기억기공간을 확보해야 한다. 즉 배렬공간을 창조하여야 한다. 배렬공간을 창조하는 문법형식은 다음과 같다.

배렬이름 = new 배렬원소형[배렬원소의 개수];

우에서 선언한 2개의 배렬은 다음과 같이 공간을 창조할수 있다.

```
MyIntarray = new int[10];
```

```
ArrayOf200Card = new D200_Card[15];
```

배렬공간을 창조하는 과정은 배렬선언과 함께 조합하여 한개의 명령문으로도 완성할수 있다. 즉

```
int MyIntArray[] = new int[10];
D200_Card[] arrayOf200Card = new D200_Card[15];
```

배렬원소형이 기본자료형인 배렬에 대해서는 배렬공간을 창조하는 동시에 매 배렬원소에 초기값을 줄수 있으며 이렇게 하면 공간창조의 new연산자를 생략할수 있다. 실례로

```
int MyIntArray[ ]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

이 명령문은 10개의 옹근수형원소배렬을 창조하고있으며 동시에 매개 원소에 초기값을 주고있다.

3) 배렬원소창조와 초기화

배렬원소가 어떤 하나의 클라스객체로 되는 배렬에 대해서는 매개 배렬원소를 창조하고 초기화하는 단계가 반드시 있어야 한다. 실례로 우의 배렬 ArrayOf200Card의 매 원소는 모두 D200_Card클라스의 객체이며 객체창조를 하는 동시에 반드시 이 객체의 구성자를 작성하여야 한다. 실례로 D200_Card클라스의 객체의 구성자는 다음과 같다.

```
D200_Card(long cn, int pw, double b, String c, double a)
{
    cardNumber = cn;
    password = pw;
    balance = b;
    connectNumber = c;
    additoryFee = a;
}
```

D200_Card클라스의 객체를 창조하려면 반드시 이 클라스의 구성자를 작성하여야 하며 상응한 실제파라메터를 주어 객체내부의 매개 속성변수나 객체를 초기화하여야 한다. 배렬 ArrayOf200Card에서의 매개 원소는 다음과 같은 순환을 리용하여 초기화할수 있다.

```
for(int i = 0; i < ArrayOf200Card.length; i++)
{
    ArrayOf200Card[i] =
        new D200_Card(200180000+i, 1111, 50.0, "200", 0.1);
}
```

이 순환을 실행한 결과 ArrayOf200Card[0]부터 ArrayOf200Card[14]까지의 모든 배렬원소에 대하여 기억공간이 확보되고 객체가 초기화된다. 여기서 length는 속성변수로서 배렬의 길이이다. 오직 모든 배렬원소의 창조와 초기화작업을 완성한 후

에야 프로그람에서 이 배렬원소를 사용할수 있다. 만일 우의 단계를 거치지 않고 초기화한 배렬을 사용하면 NullPointException의 례외오유를 나타낼수 있다.

2.5.2. 벡토르

벡토르(Vector)는 java.util패키지(java.util패키지는 각종 상용도구클라스들을 전문적으로 보존하는 클라스서고이다.)가 제공하는 도구클라스로서 배렬과 마찬가지로 순서를 가지고 자료를 저축하는 자료구조이다. 그러나 배렬에 비하여 보다 강한 기능을 가지고있다. 그것은 서로 다른 형들의 원소가 공존하는 가변길이의 배렬을 리용한다는것이다. 매개 Vector클라스의 객체는 하나의 완전한 자료(매개 자료는 어떤 형의 객체를 리용하여 표현한다.)순서렬로 표현할수 있다.

1) 벡토르클라스의 객체창조

Vector클라스는 3개의 구성자를 가지는데 여기에서 가장 복잡한 구성자를 소개한다.

```
public Vector(int initCapacity, int capacityIncrement);
```

이 구성자는 두개의 형식파라메터를 가진다. initCapacity는 창조시 Vector순서렬이 포함하는 원소개수를 표시하며 capacityIncrement는 만일 Vector순서렬에 원소를 추가하는 경우 한번에 몇개를 추가하겠는가를 나타내는 파라메터이다. 아래의 명령문은 이 구성자를 리용하여 벡토르순서렬을 창조한다.

```
Vector MyVector = new Vector(100,50);
```

이 명령문은 창조하려는 MyVector벡토르순서렬이 초기에 100개 원소의 공간을 가지며 이후에 일단 사용하면 단위가 50으로 증가되여 순서렬의 개수는 150, 200, …으로 변화된다. Vector순서렬의 창조시 순서렬에서 원소의 형을 지정할 필요가 없고 사용시에 확정하면 된다.

2) 벡토르순서렬에서 원소의 추가

벡토르순서렬에 원소를 추가하는 메쏘드에는 2가지가 있다. addElement( )메쏘드는 새 원소를 벡토르순서렬의 마지막부분에 추가하며 insertElement( )메쏘드는 새 원소를 순서렬의 지정위치에 추가한다.

```
addElement(Object obj);
insertElement(Object obj, int index);
```

여기서 obj는 벡토르순서렬에 추가하는 객체이며 index는 추가되는 위치(0은 첫째 위치를 의미함)이다.

아래에 두 메쏘드를 사용하는 례제를 소개한다.

```
Vector Myvector = new Vector( );
for(int i =1; i <= 10; i++)
{
MyVector.addElement(new D200_Card(200180000+i,1111,50.0,"200",0.1));
}
MyVector.insertElement(new IP_Card(12345678,1234,100.0,"200"),0);
```

이 프로그람은 우선 빈 Vector객체를 창조하고 다음에 10개의 D200_Card클라스의 객체들을 각각 창조하여 차례로 벡토르순서렬의 마지막에 추가하고 다시 IP_Card클라스의 객체를 벡토르순서렬의 제일 앞에 추가한다. 이로부터 알수 있는바와 같이 Vector순서렬에서의 원소는 서로 다른 형일수 있다.

3) 벡토르순서렬에서 원소의 수정, 삭제

아래의 메쏘드들을 사용하여 벡토르순서렬의 원소를 수정삭제할수 있다.

· void setElementAt(Object obj, int index)

벡토르순서렬의 index위치에 있는 객체원소를 obj로 설정하며 만일 이 위치에 원래의 원소가 있으면 덧태운다.

· boolean removeElement(object obj)

벡토르순서렬에서 지정한 obj객체와 같은 첫번째 원소를 삭제하고 동시에 뒤의 원소들을 앞으로 이동한다.

· void removeAllElements( )

벡토르순서렬의 모든 원소들을 제거한다.

아래의 례제에서는 먼저 Vector를 창조하고 다음에 문자렬객체 to를 모두 삭제한다.

```
Vector MyVector = new Vector(100);
for(int i = 0; i < 10; i++)
{
    MyVector.addElement("welcome");
    MyVector.addElement("to");
    MyVector.addElement("Pyongyang");
}
while(MyVector.removeElement("to"));
```

만일 벡토르순서렬에서 삭제하려는 객체가 존재하지 않으면 removeElement( )메쏘드는 false를 귀환시킨다. 우의 프로그람은 이 특징을 리용하여 원래의 순서렬에서 모든 to객체를 삭제하였다.

4) 벡토르순서렬에서 원소의 검색

벡토르순서렬에서 어떤 원소를 검색하는데 리용되는 메쏘드는 아래와 같다.

· Object elementAt(int index)

먼저 지정된 위치에 있는 원소를 검색한다. 검색된것은 Object형의 객체이므로 사용하기전에 강제형변환을 진행하여 어떤 구체적인 하위클라스의 객체로 변환한다. 실례로 아래의 순서렬의 첫번째 원소는 한개의 문자렬이다.

```
String str = (String)MyVector.elementAt(0);
```

· boolean contains(Object obj)

벡토르순서렬에서 지정한 객체원소 obj를 포함하는가 안하는가를 검색하여 포함하면 true를 귀환시키고 그렇지 않으면 false를 귀환시킨다.

· int indexOf(Object obj, int start_index);

지정한 start_index위치로부터 뒤방향으로 탐색을 시작하여 obj와 같은 첫번째 지정객체의 첨수값을 귀환시킨다. 만일 지정객체가 존재하지 않으면 -1을 귀환시킨다.

아래의 프로그람은 벡토르순서렬에서 Welcome인 모든 원소를 검색하고 그것들의 위치를 출력한다.

```
int i = 0;
while((i = MyVector.indexOf("Welcome",i)) != -1)
    System.out.println(i);
```

Vector를 사용할 때 한가지 특별히 주의해야 할 문제는 먼저 창조한 후에 사용해야 한다는것이다. 만일 new를 먼저 사용하지 않고 구성자를 리용하여 Vector클라스의 객체를 창조하고 addElement( )등의 메쏘드를 직접 사용하면 탄창넘침이나 null지적자의 사용 등의 례외현상을 일으켜 프로그람의 정상운영을 방해할수 있다.

제6절. 문자렬

- 문자렬은 상수인가 변수인가에 관계없이 클라스의 객체를 사용하여 실현한다.
- 문자렬처리와 관련한 클라스: String, StringBuffer

문자렬은 프로그람작성에서 늘 사용해야 할 자료구조이다. 이것은 문자의 순서렬로서 어떤 측면에서 보면 문자들의 배렬과 류사하다. 실제상 어떤 프로그람언어(C언어)들에서 문자렬은 문자배렬을 사용하여 실현한다. 그러나 Java객체지향언어에서의 문자렬은 상수인가 변수인가에 관계없이 클라스의 객체를 사용하여 실현한다.

프로그람에서 쓰는 문자렬을 2가지로 구분할수 있다. 하나는 창조후에 다시 수정변경할수 없는 문자렬상수이며 다른 하나는 창조후에 수정변경이 가능한 문자렬변수이다. Java에서 문자렬상수를 보관하는 객체는 String클라스에 속한다. 문자렬변수에 대하여서는 프로그람에서 추가, 삽입, 수정 등의 클라스조작을 하여야 하므로 일반적으로 StringBuffer클라스의 객체에 보관해놓는다.

2.6.1. String클라스

문자렬상수는 String클라스의 객체를 사용하여 표시한다. 문자렬상수는 단인용부호를 리용하는 단일문자이며 'a', '\n' 등으로 표시한다. 한편 문자렬상수는 쌍인용부호를 리용하는 문자순서렬이다.(례하면 "a", "\n", "Hello" 등) C언어에서의 문자렬은 문자배렬로 구성되여 매개 문자렬의 마지막은 "\0"으로 표시한다. 그러나 Java의 문자렬상수는 보통 String클라스의 객체가 존재하여 전문적인 속성을 가지고 그의 길이를 규정한다. 이 절에서는 문자렬상수를 보관해놓는 String클라스에 대하여 String객체의 창조, 사용, 조작을 서술한다.

1) 문자렬상수 String객체의 창조

String클라스의 객체가 표시하는것은 문자렬상수이므로 일반적으로 String문자렬이 창조되면 그의 길이와 내용은 다시 변경할수 없다. 그러므로 String객체를 창조할때 보통 String클라스의 구성자에 파라메터를 설정하여 문자렬의 내용을 지정하여야 한다. 아래에 String클라스의 구성자와 그의 사용메쏘드를 간단히 서술한다.

· public String( )

이 구성자는 한개의 빈 문자렬상수를 창조하는데 쓰인다.

· public String(String value)

이 구성자는 이미 존재하는 문자렬상수를 리용하여 새로운 String객체를 창조한다. 이때 이 객체의 내용은 주어진 문자렬상수와 일치한다. 이 문자렬상수는 다른 String객체일수도 있고 쌍인용부호를 리용하여 표현한 직접적인 상수일수도 있다.

· public Stirng(StringBuffer buffer)

이 구성자는 이미 있는 문자배렬의 내용을 리용하여 새로 만들려는 String객체를 초기화한다. StringBuffer객체는 내용, 길이가 변경될수 있는 문자렬변수를 의미한다.

· public String(char value[])

이 구성자는 이미 있는 문자배렬을 리용하여 새로 창조하려는 String객체를 초기화한다. 실례로

```
String s;
```

이때 s의 값은 null이며 s를 사용하려면 반드시 이것에 기억기공간을 확보하여야 한다.

```
s = new String("ABC");
```

이렇게 두번째 구성자의 리용을 통하여 문자렬 s는 "ABC"로 설정된다. 우의 2개 명령문을 1개 명령문으로 쓸수도 있다.

```
String s = new String("ABC");
```

Java에는 자주 쓰이는 String객체창조방법이 있다. 이 방법은 쌍인용부호를 리용한 문자렬상수를 새로 창조하는 String객체에 직접 값주기하는것이다. 즉

```
String s = "ABC";
```

2) 문자렬상수의 조작

String클라스에는 문자렬상수에 대한 조작을 진행하는 메쏘드들이 많다. 구체적인 메쏘드는 아래와 같다.

```
public int length( );
```

이 메쏘드는 문자렬객체에서 문자의 개수를 얻는데 리용한다. 실례로 아래의 코드를 실행해보자.

```
String s = "Hello!";
System.out.println(s.length( ));
```

화면에서는 6을 현시하며 따라서 문자렬 《Hello!》의 길이는 6이다. 주의해야 할 것은 Java에서 매개 문자는 16bit의 Unicode문자이므로 조선어나 한자는 영문이나 다른 기호와 마찬가지로 한개 문자만을 사용하여 표시한다. 만일 우의 명령문에서 문자렬을 《안녕하십니까》로 바꾸어도 문자렬의 길이는 여전히 6이다.

3) 문자렬의 앞붙이와 뒤붙이 판단

```
public boolean startsWith(String prefix);
public boolean endsWith(String suffix);
```

이 2가지 메쏘드로 각각 현재 문자렬이 앞붙이인가 뒤붙이인가를 판단할수 있다.

문자렬을 구분하는 앞붙이와 뒤붙이는 어떤 경우에는 아주 필요하다. 실례로 전화국의 구사용자의 전화번호는 문자부분렬 6278로 시작하고 신사용자의 전화번호는 8278로 시작한다고 하자. 만일 전화국이 구사용자와 신사용자를 구분하려면 아래의 명령문을 리용할수 있다. 즉

```
String s = User.getPhone( ); //User는 전화국사용자객체이다.
if(s.startsWith("6278")) //전화번호가 "6278"로 시작하는가를 판단
{
    …
}
```

또한 실례로 거주자신분증번호의 마지막 한개의 수자가 거주자의 성별(기수이면 남성, 우수이면 녀성)을 나타낸다고 하자. 이때 String객체 s가 어떤 거주자의 신분증번호라고 하면 아래의 명령문은 그의 성별을 판단할수 있다.

```
If(s.endsWith("0") | | s.endsWith("2") | | s.endsWith("4")
| |s.endsWith("6") | | s.endsWith("8"))
{
    System.out.println("이 사람은 녀성이다.");
}
```

startsWith와 endsWith메쏘드의 우점은 판단하는 앞붙이, 뒤붙이의 길이를 제한하지 않는다는것이다. 앞의 례제에서 판단해야 할 앞붙이가 6278로부터 627로 변해도 원래 메쏘드는 여전히 유효하며 프로그람을 수정할 필요가 없다.

4) 문자렬에서 단일문자의 검색

```
public int indexOf(int ch);
public int indexOf(int ch, int fromIndex);
```

우에서 서술한 2개의 메쏘드는 현재문자렬에서 어떤 특정한 문자가 출현하는 위치를 검색한다. 첫번째 메쏘드는 문자 ch가 현재문자렬에서 처음으로 출현하는 위치를 앞에서부터 뒤방향으로 검색하며 문자 ch가 출현하는 위치를 귀환시킨다. 만일 찾을수 없으면 -1을 귀환시킨다. 다음의 코드는 주어진 문자렬에서 J가 출현하는 첫 위치인 0을 옹근수형변수 idx에 값주기한다.

```
String s = "Java는 객체지향언어이며 JavaScript는 스크립트언어이다";
int idx = s.indexOf((int)'J');
```

두번째 메쏘드는 문자 ch를 검색할 때 현재문자렬에서 fromIndex위치의 문자로부터 뒤방향으로 검색하여 이 문자가 처음 출현하는 위치를 귀환시킨다. 아래의 명령문은 문자렬에서 지정문자의 모든 출현위치를 검색한다.

```
String s = "Java는 객체지향언어이며 JavaScript는 스크립트언어이다";
int i = -1;
do{
    i = s.indexOf((int)'a',i + 1);
    System.out.print(i + "\t");
}while(i != -1);
```

실행결과는 1　　3　　16　　18　　-1이다.

아래의 2개 메쏘드 역시 문자렬에서 한개 문자를 검색하는 메쏘드이며 다른점은 이 메쏘드들이 문자렬의 마지막에서부터 앞방향으로 검색한다는것이다.

```
public int lastIndexOf(int ch);
public int lastIndexOf(int ch, int fromIndex);
```

5) 문자렬에서 부분렬의 검색

문자렬에서 문자부분렬을 검색하는것은 문자렬에서 한개 문자를 검색하는것과 류사하며 역시 4개의 메쏘드들을 가지고있다. 그것은 한개 문자를 검색하는 4개의 메쏘드에서 지정문자 ch를 지정문자부분렬 str로 바꾼것이다. 아래의 례제는 문자렬마지막부터 앞방향으로 모든 부분렬이 출현하는 위치를 순차적으로 검색하고있다.

```
String s = "Java는 객체지향언어이며 JavaScript는 스크립트언어이다";
String sub = "언어";
for(int i = s.length( ); i != -1;)
{
    i = s.lastIndexOf(sub,i - 1);
    System.out.print(i + "\t");
}
```

우의 프로그람실행결과는 31　　10　　-1이다.

그밖에 문자렬에서 어떤 문자를 얻는 메쏘드도 있다.

```
public char charAt(int index);
```

이 메쏘드는 한개 문자렬에서 index로 지정한 위치에 있는 문자를 얻고 이 문자를 귀환시킨다.(index는 0부터 계산한다)

6) 두개 문자렬의 비교

```
public int compareTo(String anotherString);
public boolean equals(Object anObject);
public boolean equalsIgnoreCase(String anotherString);
```

이 메쏘드들은 2개의 문자렬이 같은가 같지 않은가를 비교하는데 리용한다. equals메쏘드는 Object클라스를 재정의하는 메쏘드로서 현재문자렬을 메쏘드의 파라메터목록에서 준 문자렬과 비교하여 두 문자렬이 같으면 true를 귀환시키고 그렇지 않으면 false를 귀환시킨다. equalsIgnoreCase메쏘드는 equals메쏘드와 류사하며 문자렬을 비교할 때 대소문자를 고려하지 않는다. 실례로 아래의 명령문에서 equals메쏘드와 equalsIgnoreCase메쏘드를 리용하여 2개 문자렬을 비교하면 첫번째 명령문에서는 대소문자를 구분하므로 비교결과는 거짓으로 되고 두번째 명령문에서는 대소문자를 구분하지 않으므로 비교결과는 참으로 된다. 즉

```
String s1 = "Hello! World", s2 = "hello! world";
boolean b1 = s1.equals(s2);
boolean b2 = s1.equalsIgnoreCase(s2);
```

문자렬을 비교하는 메쏘드에는 compareTo( )도 있는데 이 메쏘드는 현재문자렬을 한개 파라메터문자렬과 서로 비교하여 옹근수값을 귀환시킨다. 만일 현재문자렬과 파라메터문자렬이 완전히 같으면 compareTo( )메쏘드는 0을 귀환하며 현재문자렬이 자모순에 따라 파라메터문자렬보다 크면 0보다 큰 옹근수값을 귀환시킨다. 반대로 현재문자렬이 자모순에 따라 파라메터문자렬보다 작으면 0보다 작은 옹근수값을 귀환시킨다. 아래에서는 3개의 문자렬을 compareTo( )메쏘드를 리용하여 비교하였다.

```
String s = "abc", s1 = "aab", s2 = "abd", s3 = "abc";
int i, j, k;
i = s.compareTo(s1);
j = s.compareTo(s2);
k = s.compareTo(s3);
System.out.print("i=" + i + "\t" + "j=" + j + "\t" + "k=" + k);
```

명령문의 집행결과는 i=1 j=-1 k=0으로 된다.

7) 문자부분렬의 련결

```
public String concat(String str);
```

이 메쏘드는 파라메터문자렬을 현재문자렬의 마지막부분에 련결하고 련결된 긴 문자렬을 귀환시킨다. 그러나 현재문자렬자체는 변하지 않는다. 아래의 례제를 보자.

```
String s = "Hello!";
System.out.println(s.concat("World!"));
System.out.println(s);
```

실행결과는 다음과 같다.

```
Hello!World! //련결후의 새로운 문자렬
Hello! //원래의 문자렬은 변하지 않았다.
```

2.6.2. StringBuffer클라스

Java에서 문자렬을 실현하는데 쓰이는 다른 하나의 클라스는 StringBuffer클라스로서 클라스의 매개 객체는 모두 확장수정할수 있는 문자렬변수이다.

1) 문자렬변수 StringBuffer객체의 창조

StringBuffer가 표시하는것은 확장가능하고 수정할수 있는 문자렬이므로 StringBuffer클라스의 객체를 창조할 때 꼭 문자렬초기값을 주어야 하는것은 아니다. StringBuffer클라스의 구성자에는 다음과 같은것들이 있다.

```
public StringBuffer( );
public StringBuffer(int length);
public Stringbuffer(String str);
```

첫번째 구성자는 빈 StringBuffer객체를 창조하며 두번째는 새로 창조하는 StringBuffer객체의 길이를 주었다. 세번째는 이미 존재하는 String객체를 리용하여 StringBuffer객체를 초기화한다. 아래의 명령문은 우의 3가지 구성자를 리용하여 문자렬을 창조하는 례제이다.

```
StringBuffer MystrBuff1 = new StringBuffer( );
StringBuffer MystrBuff2 = new StringBuffer(s);
StringBuffer MystrBuff3 = new StringBuffer("Hello,Guys!");
```

주의해야 할것은 첫번째 객체 MyStrBuff1은 대응하는 기억단위를 가지지 않는다는것이며 확장한 후에야 사용할수 있다는것이다.

2) 문자렬변수의 확장, 수정, 조작

StringBuffer클라스에는 포함된 문자를 확장하는데 쓰이는 아래와 같은 2가지 메쏘드가 있다.

```
public stringBuffer append(파라메터객체형 파라메터객체이름);
public stringBuffer insert(int 삽입위치, 파라메터객체형 파라메터객체이름);
```

append메쏘드는 지정한 파라메터객체를 문자렬로 변환하여 원래의 StringBuffer 문자렬객체뒤에 추가한다. 한편 insert메쏘드는 지정한 위치에 파라메터객체를 삽입한다. 추가하거나 삽입하는 파라메터객체는 여러가지 자료형의 자료일수 있다.(례: int, double, char, String 등) 아래의 례제를 고찰하자.

```
StringBuffer MyStrBuff1 = new StringBuffer( );
MyStrBuff1.append("Hello, Guys!");
System.out.println(MyStrBuff1.toString( ));
MyStrBuff1.insert(6,30);
System.out.println(MyStrBuff1.toString( ));
```

우의 프로그람실행결과는 다음과 같다.

```
Hello, Guys!
Hello,30 Guys!
```

주의하여야 할것은 StringBuffer를 화면상에서 현시하자면 반드시 toString메쏘드를 리용하여 그것을 문자렬상수로 변환하여야 하며 때문에 printStream의 메쏘드 println()은 StringBuffer형의 파라메터를 접수할수 없다.

StringBuffer에는 또한 문자렬을 수정하는데 비교적 유용한 메쏘드가 있다.

```
public void setCharAt(int index,char ch);
```

이 메쏘드는 지정위치에 있는 문자를 지정한 다른 문자로 바꿀수 있다. 실례로 아래의 코드는 goat인 문자렬을 coat로 변환한다. 즉

```
StringBuffer MyStrBuff = new StringBuffer("goat");
MyStrBuff.setCharAt(0,'c');
```

3) 문자렬의 값주기와 더하기

문자렬은 자주 사용하는 자료형이므로 프로그람작성을 편리하게 하기 위하여 Java번역체계에서는 문자렬의 더하기와 값주기를 도입하였다. 아래에 실례를 준다.

```
String MyStr = "Hello,";
MyStr = MyStr + "Guys!";
```

이 2개의 명령문은 얼핏 보기에는 틀린것 같이 보인다. 그러나 실제상 그것들은 문법규칙에 부합되며 각각

```
String MyStr = new StringBuffer( ).append("Hello").toString( );
MyStr = new StringBuffer( ).append(MyStr).append("Guys!").toString( );
```

과 같다.

문자렬의 값주기와 더하기는 아주 편리하고 실용적이므로 실제적인 프로그람작성에서 많이 쓰인다.

2.6.3. Java Application지령행파라메터

Java Application은 지령행을 리용하여 기동실행하므로 지령행파라메터는 Java Application에 자료를 전달하는 유효한 수단으로 된다. 실례 2-11을 통하여 지령행파라메터를 어떻게 사용하는가를 고찰한다.

실례 2-11

```
Example 2-11 UseComLParameter.java
1: public class UseComLParameter
2: {
3:      public static void main( String args[] )
4:      {
5:          int a1, a2, a3 ;
6:          if ( args.length < 2 )
7:          {
8:              System.out.println("이 프로그람을 실행하자면 두개의
```

```
                                        지령행파라메터를 주어야 합니다.");
9:          System.exit(0);
10:       }
11:       a1 = Integer.parseInt( args[0] ) ;
12:       a2 = Integer.parseInt( args[1] ) ;
13:       a3 = a1 * a2 ;
14:       System.out.println(a1 + "와 " + a2 + "을 곱한 적:" + a3 );
15:   }
16:}
```

프로그람설명

실례 2-11은 지령행으로부터 두개의 옹근수를 읽어들여 그것들을 서로 곱한 후에 출력한다. 우의 프로그람을 번역한 다음 실행할 때의 지령행은 아래의 형식을 취하여야 한다.

java UseComLParameter 5 7

여기서 UseComLParameter는 실행하려는 바이트코드파일이름(즉 클라스이름)이며 5와 7은 각각 두개의 지령행파라메터이다.

보는바와 같이 Java의 지령행파라메터는 주클라스이름 다음에 있으며 파라메터들사이는 공백을 리용하여 가른다. 그렇지 않으면 쌍인용부호를 리용해서 파라메터를 구별할수도 있다. 실례로 《adog》는 하나의 지령행파라메터이다.

Java Application프로그람에서 지령행파라메터를 접수하는데 쓰이는 자료구조는 main( )메쏘드의 파라메터 args[ ]이며 이 파라메터는 문자렬배렬이다. 입력한 지령행파라메터에 따라 배렬 args[ ]는 원소를 가진다. 실례 2-11에서 6-8행은 배렬 args의 마당 length를 리용하여 사용자가 몇개의 지령행파라메터를 입력하였는가를 판단하며 만일 사용자가 입력한 지령행파라메터의 수가 프로그람요구에 부합되지 않으면 통보문을 출력하고 프로그람의 실행에서 탈퇴한다. 11행에서 배렬원소 args[0]은 첫번째 지령행파라메터를 접수하는데 쓰인다. 우의 실례에서는 5로 된다. 12행에서 arg[1]은 두번째 지령행파라메터를 접수하는데 쓰인다. 우의 실례에서는 7이다.

주의할것은 모든 지령행파라메터가 모두 문자렬 String형의 객체이므로 만일 파라메터를 다른 형의 자료로서 사용하자면 상응한 형변환을 하여야 한다. 우의 실례에서 11행과 12행은 Integer클라스의 정적메쏘드 parseInt를 사용하여 String을 옹근수형 int로 형변환하였다. 프로그람실행결과는 그림 2-13과 같다.

```
D:\Javatextbook\Test>java UseComLParameter 5 7
5와 7을 곱한 적:35

D:\Javatextbook\Test>java UseComLParameter 5
이 프로그람을 실행하자면 두개의 지령행파라메터를 주어야 합니다.

D:\Javatextbook\Test>
```

그림 2-13. 실례 2-11의 실행결과

제3장. 추상과 내장, 클라스

이 장에서는 대상지향프로그람설계에서의 중요특징들인 추상과 내장에 대하여 고찰한다. 또한 Java에서 클라스와 객체를 정의하고 장식부, 구성자의 구체적인 사용규칙들을 서술한다.

제1절. 추상과 내장

- **추상은 구체적인 사물을 추상적인 개념의 범주로 묶은것이다.**
- **내장은 자료와 조작을 객체의 내부에 숨기고 어떤 인증조작을 통해서만 객체와 교류하는것이다.**

3.1.1. 추상

추상은 과학연구에서 늘 사용하는 한가지 방법이다. 즉 연구대상에 무관계한 부분이나 일시적으로 고려하지 않는 부분을 제거하고 연구사업과 련관이 있는 실질적인 내용만을 골라내여 고찰하는 방법론이다. 콤퓨터기술의 쏘프트웨어개발방법에서 사용하는 추상에는 2가지 류형이 있다. 하나는 수속추상이고 다른 하나는 자료추상이다.

수속추상은 전체 체계의 기능을 몇개의 부분으로 나누고 기능을 완성하는 과정과 단계이다. 수속추상을 사용하는것은 전체 프로그람의 복잡도를 조종하고 낮추는데 유리하다. 그러나 이 방법자체가 자유도가 크고 규칙과 표준화가 어려우며 조작하는데 일정한 결함이 있고 품질보증이 쉽지 않다.

자료추상은 수속추상과 다른 추상방법으로서 체계에서 처리하여야 할 자료와 이 자료들사이의 조작들을 서로 결합시킨다. 기능, 성질, 작용 등의 인자에 따라 추상은 서로 다른 추상자료형들로 된다. 매개 추상자료형은 자료를 포함하기도 하고 이 자료들에 대한 인증조작을 포함하기도 한다.

객체지향쏘프트웨어개발방법의 주요특징의 하나는 바로 자료추상방법을 리용하여 프로그람의 클라스, 객체, 메쏘드를 구성하는것이다. 객체지향기술에서 이 자료추상방법을 리용할수 있다. 한 측면은 핵심문제와 련관이 없는 부분을 제거하고 개발작업을 비교적 관건적이고 주요한 부분에 집중시키며 다른 측면은 자료추상과정에서 자료와 조작에 대한 분석, 판별과 정의에 대해 개발자들이 정확한 인식을 가지도록 방조한다는데 있다. 추상의 형성과정은 설계와 프로그람작성의 기초 및 담보로 된다.

실례로 은행의 일상업무와 관련한 문제를 처리하는데서 가장 핵심적인 문제는 모든 자금, 장부, 거래이다. 이 핵심문제와 련관이 있는 조작에 근거하여 저금, 송금, 대부, 채권과 조작들이 처리하는 자료(례: 금액, 구좌번호, 날자 등)들을 포괄시키면 고찰의 중점으로 되는 구좌를 표시하는 추상자료형을 세울수 있다. 그러나 은행의 기타업무와 일상작업(례: 감시, 안전경계, 저금봉사항목)들은 제외된다. 반대로 은행봉

사수준과 작업효률을 높이는 종합관리체계이면 첫번째 체계에 의하여 홀시되였던 몇가지 작업들이 두번째 체계의 추상자료형의 일부분으로 될수도 있다. 이렇게 추상은 사람들이 작업의 중점을 명확히 할수 있게 한다.

3.1.2. 내장

객체지향방법의 내장특성은 추상특성과 밀접한 련관이 있는 특성이다. 구체적으로 내장은 추상자료형을 리용하여 자료와 자료에 기초한 조작을 함께 매몰한다는것을 의미한다. 자료는 추상자료형의 내부에 보존되며 체계의 기타 부분들은 자료밖에서 인증조작을 통하여서만 이 추상자료형과 교류하고 호상작용할수 있다.

제2절. 클라스

- 클라스서고의 원만한 사용은 프로그람작성효률과 품질을 높이는데서 필수적인것이다.
- 클라스의 정의는 머리부와 본체부분으로 구성된다.
- 객체는 반드시 클라스를 통해서만 정의한다.

추상과 내장은 클라스의 정의와 사용측면에서 주로 표현된다. 이 절에서는 Java에서 클라스를 어떻게 정의하는가를 고찰한다.

3.2.1. 체계가 정의하는 클라스

실례 3-1

```
Example 3-1 PhoneCard.java
1: class PhoneCard
2: {
3:      long cardNumber;
4:      private int password;
5:      double balance;
6:      String connectNumber;
7:      boolean connected;
8:
9:      boolean performConnection(long cn, int pw)
10:     {
11:         if(cn == cardNumber && pw == password)
```

```
12:         {
13:             connected = true;
14:             return true;
15:         }
16:         else
17:         {
18:             connected = false;
19:             return false;
20:         }
21:     }
22:     double getBalance( )
23:     {
24:         if(connected)
25:             return balance;
26:         else
27:             return -1;
28:     }
29:     void performDial( )
30:     {
31:         if(connected)
32:             balance -= 0.5;
33:     }
34:}
```

프로그람설명

실례 3-1의 프로그람은 사용자클라스 PhoneCard를 정의한다. 1행은 클라스머리부를 정의하고 2-34행은 클라스본체를 정의한다.

PhoneCard클라스는 전화카드를 추상한 클라스이다. 실례에서는 PhoneCard클라스에 대한 5개의 마당과 3개의 메쏘드를 정의하고있다. 여기서 CardNumber마당은 긴 옹근수형변수이며 전화카드의 카드번호를 나타낸다. password마당은 옹근수형변수이며 전화카드의 비밀번호를 나타낸다. balance는 배정확도형변수로서 전화카드에 남아있는 금액을 나타낸다. connectNumber마당은 문자렬객체로서 전화카드에 대한 접수전화번호를 나타낸다.(례: 200전화카드의 접수번호는 200이며 교내201전화카드의 접수번호는 201이다.) Connected마당은 론리형변수로서 전화가 통화인가 아닌가를 나타낸다. performConnection( )메쏘드는 전화를 접수하는 조작을 실현한다. 만일 사용자가 돌린 카드번호와 비밀번호가 전화카드안에 보존된 카드번호, 비밀번호와 일치하면 전화가 통화된다. getBalance( )메쏘드는 우선 전화가 통화인가를 검사하고 통화이면 카드안에 남아있는 금액을 귀환시킨다. 그렇지 않으면 수값 -1을 귀환한다.

performDial( )메쏘드 역시 우선 전화가 통화인가를 검사하고 통화이면 한번의 통화료금 0.5원을 공제한다.

3.2.2. 객체창조와 구성자의 정의

1) 객체창조

Java프로그람에서 클라스를 정의하는 최종목적은 그것을 사용하자는것이다. 여기에서는 클라스의 객체를 어떻게 창조하는가를 고찰한다.

앞의 실례들에서 도형사용자대면부의 입출력기능을 완성하기 위하여 몇가지 체계클라스의 객체를 창조하였다. 명령문은 다음과 같다.

```
TextField input = new TextField(6);
```

이것은 java.awt패키지의 체계클라스 TextField에서 이름이 input인 객체를 창조한다. 마찬가지로 실례 3-1에서 정의한 PhoneCard클라스의 객체창조도 아래와 같이 할수 있다.

```
PhoneCard myCard = new phoneCard( );
```

객체창조의 일반형식은 아래와 같다.

클라스이름 새로 창조하는 객체이름 = new 구성자();

객체창조는 변수선언과 류사하며 우선 새롭게 창조되는 객체가 속하는 클라스이름을 쓰고 다음 새롭게 창조되는 객체의 이름을 쓴다. 값주기기호 오른변의 new는 새로 창조되는 객체에 대하여 기억기공간을 확보하는 연산자이다. 이것을 집행하면 객체에 대하여 기억기공간을 확보하고 마당과 메쏘드를 보존한다.

PhoneCard클라스에서는 5개 마당과 3개 메쏘드를 정의하고있다. 매 구체적인 phoneCard클라스의 객체(례: 우의 myCard객체)의 기억기공간은 5개의 마당, 3개의 메쏘드를 보존하며 myCard객체의 마당과 메쏘드는 각각 myCard.cardNumber, myCard.password, myCard.balance, myCard.connectNumber, myCard.connected, myCard.performConnection(), myCard.getBalance( ), myCard.performDial( )이다. 이 마당과 메쏘드는 하나의 기억기내에 보존되며 이 기억기는 myCard객체가 차지하고있는 기억기이다. 만일 phoneCard클라스의 또 다른 객체 newCard를 창조하면 newCard객체는 기억기내에서 다른 phoneCard객체와 서로 무관계한 마당과 메쏘드를 쓰게 되며 자기의 메쏘드에 따라 자기의 마당을 관리한다. 이것은 객체지향에서 내장특성의 표현이다.

객체의 마당이나 메쏘드에 접근하거나 호출하려면 우선 이 객체에 접근하고 다음 연산자 《.》을 리용하여 이 객체의 어떤 마당이나 메쏘드에 련결되여야 한다. 실례로 myCard객체의 balance마당을 50으로 설정하려면 아래의 명령문을 사용할수 있다.

```
myCard.balance = 50;
```

2) 구성자

객체창조가 변수선언과 다른 점의 하나는 객체를 창조하는 동시에 이 객체의 구성자를 리용하여 객체의 초기화작업을 진행한다는것이다. 변수선언시에는 값주기문을 리용하여 그에 초기값을 주는데 한개 객체는 몇개의 마당을 포함하며 몇개의 값주기문을 요구하므로 객체창조시에 몇개의 초기값주기명령문을 하나의 메쏘드로 구성하여 동시에 집행하게 되는데 이 메쏘드가 바로 구성자이다.

구성자는 클라스와 같은 이름을 가지는 메쏘드이다. new연산자를 리용하여 창조되는 객체의 기억기공간을 확보한 후 구성자를 리용하여 새로 창조한 객체를 초기화한다. 구성자는 클라스의 일종의 특수한 메쏘드이다.

구성자의 특징:

① 구성자의 이름과 클라스이름은 서로 같다.

② 구성자는 귀환형을 가지지 않는다.

③ 구성자의 주요작용은 클라스객체에 대한 초기화를 완성하는것이다.

④ 구성자는 프로그람작성자에 의하여 직접 호출될수 없다.

클라스의 객체의 창조는 이 클라스의 구성자를 자동적으로 호출하여 새 객체에 대한 초기화를 진행한다. 실례로 PhoneCard클라스에 대하여 아래와 같은 구성자를 정의하고 그의 몇개 마당을 초기화할수 있다.

```
PhoneCard(long cn, int pw, double b, String s)
{
    cardNumber = cn;
    password = pw;
    balance = b;
    connectNumber = s;
    connected = false;
}
```

여기에서 매개 마당은 새로 창조하는 객체의 마당이므로 객체이름을 앞붙이로 사용할 필요가 없다. 구성자를 정의한 후에 아래의 명령문을 사용하여 PhoneCard객체를 창조하고 초기화할수 있다.

```
PhoneCard newCard = new PhoneCard(12345678,1234,50.0,"300");
```

이 객체의 카드번호는 12345678이며 비밀번호는 1234이고 금액은 50.0, 전화카드의 접수번호는 문자렬 300이다.

보는바와 같이 구성자에서는 몇개의 형식파라메터를 정의하였다.

객체를 창조하는 명령문은 구성자를 호출할 때 형과 순서가 일치하는 몇개의 실제파라메터를 제공하여야 하며 새로 창조되는 객체에 대하여 매 마당의 초기값을 지정하여야 한다. 이것을 리용하면 서로 다른 초기특성의 동일한 객체를 창조할수 있다. 앞에서 TextField객체인 input를 창조하는 명령문에서 수자 6은 구성자의 실제파라메터이며 새로 창조한 본문마당의 길이를 지정한다.

구성자는 또한 값주기이외의 다른 조작도 할수 있다. 실례로 아래와 같이 수정하면 PhoneCard메쏘드의 구성자는 실제파라메터가 제공하는 금액이 0보다 큰가를 검사하고 그렇다면 정상 값주기하며 그렇지 않으면 비법수값이라는것을 의미하므로 System.exit()메쏘드를 호출하여 조작을 탈퇴한다.

```
PhoneCard(long cn, int pw, double b, String s)
{
    cardNumber = cn;
    password = pw;
    if(b > 0)
        balance = b;
    else
        System.exit(1);
    connectNumber =  s;
    connected = false;
}
```

만일 사용자정의클라스가 구성자를 정의하지 않는다면 체계는 이 클라스에 대하여 기정으로 빈 구성자를 정의하는데 이것은 형식파라메터도 임의의 구체적인 명령문도 가지지 않으며 임의의 조작도 완성할수 없다.(실례 3-1에서 빈 구성자를 리용)

실례 3-2는 PhoneCard클라스를 사용하는 실례이다.

실례 3-2

```
Example 3-2 UsePhoneCard
1: public class UsePhoneCard
2: {
3:      public static void main(String args[])
4:      {
5:          PhoneCard myCard = new PhoneCard(12345678, 1234, 50.0, "300");
6:          System.out.println(myCard.toString( ));
7:      }
8: }
9: class PhoneCard
10:{
11:     long cardNumber;
12:     private int password;
13:     double balance;
14:     String connectNumber;
15:     boolean connected;
16:
```

```
17:    PhoneCard(long cn, int pw, double b, String s)
18:    {
19:        cardNumber = cn;
20:        password = pw;
21:        if(b > 0)
22:            balance = b;
23:        else
24:            System.exit(1);
25:        connectNumber = s;
26:        connected = false;
27:    }
28:    boolean performConnection(long cn, int pw)
29:    {
30:        if(cn == cardNumber && pw == password)
31:        {
32:            connected = true;
33:            return true;
34:        }
35:        else
36:        {
37:            connected = false;
38:            return false;
39:        }
40:    }
41:    double getBalnce( )
42:    {
43:        if(connected)
44:            return balance;
45:        else
46:            return -1;
47:    }
48:    void performDial( )
49:    {
50:        if(connected)
51:            balance -= 0.5;
52:    }
53:    public String toString( )
54:    {
55:        String s = "전화카드접수번호:" + connectNumber + "\n전화카드번호:"
```

```
                                          + cardNumber + "\n전화카드비밀번호:"
                                          + password + "\n남은 금액:"+balance;
56:        if(connected)
57:            return(s + "\n전화는 이미 통화입니다.");
58:        else
59:            return(s + "\n전화는 통화되지 않습니다.");
60:    }
61:}
```

실례 3-2의 5행은 PhoneCard의 객체 myCard를 창조하며 6행은 myCard의 toString( )메쏘드를 호출하여 myCard의 매 마당자료들을 하나의 정보로 조합하여 화면상에 출력한다.

그림 3-1은 실례 3-2의 실행결과이다.

```
D:\Javatextbook\Test>java UsePhoneCard
전화카드접수번호:300
전화카드번호:12345678
전화카드비밀번호:1234
남은 금액:50.0
전화는 통화되지 않습니다.

D:\Javatextbook\Test>
```

그림 3-1. 실례 3-2의 실행결과

제3절. 클라스의 장식부

- 클라스의 장식부: 접근조종부와 비접근조종부
- abstract클라스는 모든 하위클라스들의 공통적인 속성들의 집합이다.
- final클라스는 하위클라스를 가질수 없다.

Java프로그람에서 클라스를 정의할 때 class예약어를 사용하는것 외에 class의 앞에 클라스의 장식부를 붙혀 제한적으로 클라스의 속성을 표현한다. 클라스의 장식부는 접근조종부와 비접근조종부로 구분한다. 이 절에서는 클라스의 비접근조종부를 고찰한다.

3.3.1. 추상클라스

abstract장식부를 리용하여 장식한 클라스를 **추상클라스**라고 한다. 추상클라스는 구체적인 객체를 가지지 않는 개념클라스이다. 실례로 《새》를 클라스라고 가정하면 그것은 하위클라스인 《비둘기》, 《제비》, 《참새》, 《백조》 등을 파생할수 있다. 그러면 실재적인 새가 존재하는가? 그것이 비둘기도 아니고 제비나 참새도 아니며 백조는 물론 아닌 즉 임의의 구체적인 새가 아니라면 추상적인 《새》가 존재하겠는가? 답은 명백하다. 존재하지 않는다.《새》는 추상적인 개념으로 존재할뿐이며 그것은 모든 새의 공통속성을 대표한다. 임의의 구체적인 새는 《새》의 특수화를 통하여 형성한 어떤 하위클라스의 객체이다. 이러한 클라스가 Java에서는 abstract클라스이다.

추상클라스가 구체적인 객체를 가지지 않을지라도 그의 정의가 무슨 작용을 하는가?《새》의 개념이 그러한 실례로 된다. 만일 다른 사람들에게 《백조》가 무엇인가를 서술해야 한다면 보통 《백조는 목이 길고 미모가 아름다운 일종의 철새이다.》라고 말할수 있으며 《제비》에 대해 말한다면 《제비는 칼로 자른것과 같은 꼬리를 가지며 처마밑에 둥지를 틀기 좋아하는 새이다.》라고 말할수 있다. 알수 있는것처럼 정의는 대방이 이미 《새》라는 전제밑에서 세워지며 나아가서 《새》가 무엇인가를 물을 때에야 구체적으로 해석하게 된다. 《새는 날개가 있고 알을 낳는 동물이다.》 그러나 시작부터 《백조》를《목이 길고 모습이 아름다우며 날개와 털이 긴 알낳이동물이다.》라고 표현할수는 없다. 이것은 사실 하나의 최량화를 통한 개념조직방식이다.

모든 새의 공통적인 특징을 추상하여《새》를 형성하는 개념을 개괄하게 된다. 이후에 구체적으로《새》를 묘사하는 처리를 할 때 새류들의 서로 다른 특수한 점을 간단히 묘사할것을 요구할뿐이지 공통적인 특징을 다시 반복할 필요가 없다. 이러한 개념조직방식은 모든 개념이 계층적으로 분명하고 간결하며 세련되여 사람들의 일상 사유습관에 부합되게 한다.

실례로 전화카드에는 자기카드, IC카드, IP카드, 200카드, 300카드, 교내201카드와 같은 많은 류형들이 있다. 서로 다른 종류의 전화카드는 각자 자기의 특징을 가진다. 실례로 자기카드와 IC카드는 카드번호와 비밀번호를 가지고있지 않으며 200카드를 사용하면 매번 통화에 0.1원의 부가비를 더 공제해야 한다. 동시에 그것들은 공통적인 특징을 가진다. 례하면 매 전화카드에는 남은 금액이 있는데 이것은 통화의 기능을 가진다는것이다. 이를 위해 모든 종류의 전화카드에 대한 공통적인 특징들을 집합시킨 추상전화카드를 아래와 같이 정의할수 있다.

```
abstract class PhoneCard
{
    double balance;
    void performDial( )
    {
        ...
    }
}
```

추상클라스는 그의 모든 하위클라스들의 공통적인 속성들의 집합이므로 추상클라스를 사용하는 우점은 바로 이 공통적인 속성들을 충분히 리용하여 프로그람을 개발하고 유지하는 효률을 높일수 있다는것이다.

3.3.2. 최종클라스

클라스가 final장식부에 의해 장식되면 이 클라스는 하위클라스를 가질수 없다. 만일 계승관계를 가지는 클라스들을 길게 늘인 나무구조로 구성하면 모든 클라스의 상위클라스는 나무뿌리이며 매개 하위클라스는 하나의 줄기이고 final로 선언한 클라스는 나무의 잎이다. 즉 그의 아래에는 하위클라스가 더는 놓일수 없다. 그림 3-2에서 전화카드의 계층적관계를 나무구조로 보여주었다. 여기서 IC카드, 200카드들은 모두 나무잎점들이다. final클라스는 하위클라스를 가지지 않는 나무잎점이다.(나무잎점이 꼭 final클라스인것은 아니다.)

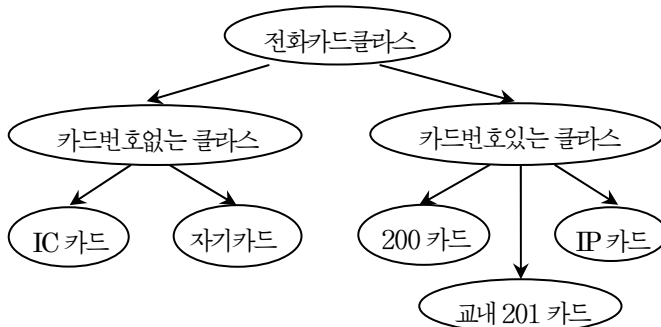



그림 3-2. 전화카드와 그의 하위클라스의 계층관계나무구조

final로 정의된 클라스는 보통 어떤 고정적인 작용을 하며 어떤 표준적인 기능을 가진다. 실례로 Java체계가 정의한 망기능을 실현하는 InetAddress, Socket 등의 클라스들은 모두 final클라스이다. Java프로그람에서 클라스이름을 통하여 클라스나 그의 객체를 인용할 때 실제적으로 인용하는것은 이 클라스나 그의 객체자체일수도 있고 또한 이 클라스의 어떤 하위클라스나 하위클라스의 객체일수도 있다. 즉 일정한 불확정성을 가지고있다. 클라스를 final로 정의하면 그것의 내용, 속성, 기능을 고정하여 그의 클라스이름으로 안정한 넘기기관계를 형성할수 있다. 하여 이 클라스를 인용할 때 실현하는 기능의 정확성을 보증할수 있다.

abstract클라스자체는 구체적인 객체를 가지지 않으며 하위클라스를 파생시킨 후 하위클라스의 객체를 다시 창조한다. 또한 사용할 메쏘드가 없으므로 의미를 가지지 않는다. 그러나 abstract와 final을 각기 다른 장식부와 합쳐 리용할수 있다. 실례로 클라스가 public abstract일수도 있고 public final일수도 있다. 여기서 public는 접근조종부이다. 하나이상의 장식부가 클라스나 클라스의 마당, 메쏘드들을 장식할 때 이 장식부들사이를 공백으로 가르며 class예약어의 앞에 쓴다. 장식부들사이의 선후배렬순서는 클라스의 성질에 아무런 영향도 미치지 않는다.

제4절. 마당과 메쏘드

- 클라스의 본체는 마당과 메쏘드로 구성된다.
- 마당은 클라스나 객체의 정적속성, 메쏘드는 동적속성이다.
- 마당의 장식부: static, final, volatile
- 메쏘드의 장식부: abstract, static, fianal, native, synchronized

3.4.1. 마당

마당(field)은 클라스와 객체의 정적속성으로서 기본수값형변수일수도 있고 다른 클라스(체계클라스나 사용자정의클라스)의 객체일수도 있다. 그러므로 클라스를 정의할 때 마당을 정의하는 조작은 변수를 선언하고 객체를 창조하는 조작이다. 클라스와 마찬가지로 마당 역시 몇개의 장식부를 가질수 있으며 접근조종부와 비접근조종부를 가진다. 여기에서는 마당의 비접근조종부를 고찰한다.

1) 정적마당

static장식부를 리용한 마당은 오직 클라스에만 속하는 **정적마당**이다.

정적마당의 본질적인 특징:

정적마당들은 클라스의 마당으로서 클라스의 구체적인 객체에 속하지 않는다. 클라스의 정적마당은 공통적인 보존단위이며 클라스의 임의의 객체가 그것에 접근할 때 취해지는 값은 모두 같다. 마찬가지로 클라스의 임의의 객체가 그것을 수정하면 다른 객체들에 대하여서도 꼭같이 수정되는것으로 된다. 아래의 프로그람부분에서는 2개의 정적마당을 정의하였다.

```
class PhoneCard200
{
    static String connectNumber = "200";
    static double additoryFee;
    long cardNumber;
    int password;
    boolean connected;
    double balance;
    …
}
```

우의 프로그람은 200전화카드에 대응하는 클라스 PhoneCard200을 정의하고있다. 모든 200전화카드의 접수번호는 200이므로 클라스를 정의하는 정적마당 connectNumber는 모든 phoneCard200객체가 공유하는 접수번호를 나타낸다. 동시에 200전화카드를 사용하여 통화하는 부가비용은 매 전화카드에 대하여 역시 일치하며 그러므로 additoryFee를 클라스의 정적마당으로 정의하였다. 아래의 프로그람은 정적마당이 클라스에서 매 객체가 공통으로 리용하는 마당이라는것을 보여준다.

실례 3-3

```
Example 3-3 TestStaticField.java
1: public class TestStaticField
2: {
3:      public static void main(String args[])
4:      {
5:          PhoneCard200 my200_1 = new PhoneCard200( );
6:          PhoneCard200 my200_2 = new PhoneCard200( );
7:          my200_1.additoryFee = 0.1;
8:          System.out.println("두번째 200카드의 부가비:" + my200_2.additoryFee);
9:          System.out.println("200카드의 부가비:" + PhoneCard200.additoryFee);
10:     }
11:}
12:class PhoneCard200
13:{
14:     static String connectNumber = "200";
15:     static double additoryFee;
16:     long cardNumber;
17:     int password;
18:     boolean connected;
19:     double balance;
20:}
```

프로그람설명

실례 3-3에서는 클라스 PhoneCard200과 주클라스 TestStaticField를 정의하였다. 5, 6행에서는 PhoneCard200클라스의 객체인 my200_1과 my200_2를 창조하고있다. 7행은 객체 my200_1을 통하여 클라스정적마당 additoryFee에 접근하며 그것에 0.1을 값주기한다. 8, 9행은 객체 my200_2와 클라스 PhoneCard200자체를 통하여 정적마당 additoryFee에 접근하며 그의 수값은 모두 객체 my200_1에 의하여 수정된 값이다. 보는바와 같이 그것들이 접근한것은 동일한 자료이다.

그림 3-3은 실례 3-3의 실행결과이다.

```
D:\Javatextbook\Test>javac TestStaticField.java

D:\Javatextbook\Test>java TestStaticField
두번째 200카드의 부가비:0.1
200카드의 부가비:0.1

D:\Javatextbook\Test>
```

그림 3-3. 실례 3-3의 실행결과

2) 정적초기화기

정적초기화기는 예약어 static와 함께 대괄호로 묶은 명령문배렬이다. 그의 작용은 구성자와 류사하게 클라스의 초기화를 완성하는것이다.

정적초기화기와 구성자의 근본적인 차이:

구성자는 새로 창조하는 매개 객체에 대한 초기화이며 정적초기화기는 클라스자체에 대한 초기화기이다.

구성자는 new연산자를 리용하여 새 객체를 창조할 때 체계에 의하여 자동적으로 집행되며 정적초기화기는 그것을 포함하는 클라스가 기억기에 적재될 때 체계에 의하여 집행된다.

구성자와 달리 정적초기화기는 메쏘드가 아니며 이름, 귀환값, 파라메터목록을 가지지 않는다.

실례 3-4에서는 정적초기화기를 사용하여 클라스를 적재할 때 클라스의 정적마당을 초기화한다.

실례 3-4

```
Example 3-4 TestStatic.java
1: public class TestStatic
2: {
3:      public static void main(String args[])
4:      {
5:          PhoneCard200 my200_1 = new PhoneCard200( );
6:          PhoneCard200 my200_2 = new PhoneCard200( );
7:          System.out.println("첫번째 200카드의 번호:" + my200_1.cardNumber);
8:          System.out.println("두번째 200카드의 번호:" + my200_2.cardNumber);
9:      }
10:}
11:class PhoneCard200
12:{
13:     static long nextCardNumber;
14:     static String connectNumber = "200";
15:     static double additoryFee;
16:     long cardNumber;
17:     int password;
18:     boolean connected;
19:     double balance;
20:
21:     static
```

```
22:   {
23:       nextCardNumber = 2001800001;
24:   }
25:   PhoneCard200( )
26:   {
27:        cardNumber = nextCardNumber++;
28:   }
29:}
```

프로그람설명

프로그람의 13행은 정적마당 nextCardNumber를 정의하며 21-24행의 정적초기화기와 25-28행의 구성자를 배합하면 PhoneCard200객체를 새로 창조하여 중복하지 않는 카드번호를 자동출력하는 기능을 완성할수 있다. PhoneCard200클라스가 기억기에 적재될 때 체계는 자동적으로 정적초기화기를 리용하여 클라스의 정적마당 nextCardNumber를 2001800001로 초기화한다. 7행에서 PhoneCard200클라스의 객체 my200_1을 창조할 때 체계는 PhoneCard200의 구성자를 리용하여 my200_1의 카드번호 cardNumber를 nextCardNumber의 현재수값으로 설정하고 다음에 nextCardNumber의 수값을 자동적으로 1만큼 증가시킨다. 8행에서 PhoneCard200의 다른 객체 my200_2를 창조할 때 체계는 다시 한번 구성자를 호출하며 이때 nextCardNumber값은 이미 2001800002로 변하였으므로 값주기후에 다시 1만큼 증가된다. 이렇게 서로 다른 PhoneCard200객체의 카드번호가 생성된다.

가령 정적초기화기의 23행을 PhoneCard의 구성자의 첫번째 행으로 이동하고 실행을 하면 어떤 결과를 얻을수 있으며 왜 그렇게 되겠는가를 알아보면 정적초기화기와 구성자의 차이를 더 잘 알수 있다.

그림 3-4는 실례 3-4의 실행결과이다.

```
D:\Javatextbook\Test>java TestStatic
첫번째 200카드의 번호:2001800001
두번째 200카드의 번호:2001800002

D:\Javatextbook\Test>
```

그림 3-4. 실례 3-4의 연산결과

3) 최종마당

프로그람에서는 일반적으로 여러가지 형의 상수 례하면 0.1, 300 등을 정의할수 있는데 그것들에 대하여 변수이름과 류사한 식별부이름을 취한다. 이렇게 하면 프로그람에서 이 이름을 써서 상수를 인용하는데 상수값을 직접 사용하는것은 아니다. final은 바로 상수를 장식하는데 쓰이는 장식부이며 만일 클라스의 마당을 final로 선

언하면 그의 값은 프로그람의 전 집행과정에서 변할수 없다. 실례로 PhoneCard200 클라스의 접수번호는 200전화카드에 대하여 고정된 문자렬 200이며 문제의 실제 상황에 따라 이 자료는 변화시키지 말아야 하므로 그것을 최종마당으로 정의할수 있다.

static **final** String ConnectNumber ="200";

final장식부를 써서 상수를 표현할 때 주의하여야 할 점들:

· 상수값의 형을 설명하여야 한다.

· 상수의 구체적인 값을 동시에 가리켜야 한다.

· 모든 클라스객체의 상수는 성원이므로 이 수값들을 모두 고정적으로 일치시키고 공간을 절약하기 위하여 상수를 보통 static로 선언한다.

4) 휘발성마당

만일 마당을 volatile장식부에 의하여 장식하면 이 마당은 동시에 몇개의 토막처리에 의해 조종수정될수 있다. 즉 이 마당은 현재의 프로그람에 의하여 장악될뿐아니라 기타 미지의 프로그람조작에 의하여 값이 변경될수 있다. 그러므로 사용할 때에 이 영향인자에 특별히 주의해야 한다. 보통 volatile은 외부입력을 접수하는 마당을 장식한다. 실례로 현재시간을 표시하는 변수는 체계의 부분토막처리에 의해 수시로 수정되며 따라서 프로그람에서는 가장 새로운 현재의 체계시간을 나타내게 된다. 이때 그것을 휘발성마당으로 정의할수 있다.

3.4.2. 메쏘드

메쏘드는 클라스의 동적속성으로서 클라스가 가지고있는 기능과 조작을 표시하며 클라스와 객체의 자료들을 함께 내장한다. Java에서 메쏘드는 다른 언어의 함수나 수속과 류사하며 어떤 조작을 완성하는 기능을 수행한다. 메쏘드는 메쏘드머리부와 메쏘드본체로 구성되며 일반형식은 다음과 같다.

```
장식부1 장식부2 … 귀환값형 메쏘드이름(형식파라메터목록) throw [례외목록]
{
    메쏘드본체의 명령문들;
}
```

여기서 형식파라메터목록의 형식은 다음과 같다.

형식파라메터형1 형식파라메터이름1, 형식파라메터형2 형식파라메터이름2, …

소괄호는 메쏘드의 표식이며 프로그람은 메쏘드이름을 리용하여 메쏘드를 호출한다. 형식파라메터는 메쏘드가 그것을 호출하는 환경으로부터 입력하는 자료이며 귀환값은 메쏘드가 조작완성된 후 그것을 호출하는 환경에 반환하는 자료이다. 메쏘드를 정의하는 목적은 상대적독립성과 상용기능을 가진 모듈을 정의하는데 있다. 이것은 프로그람구조를 뚜렷하게 하고 또한 서로 다른 경우에 반복리용하는데 유리하다. 아래의 실례 3-5는 실례 2-10의 완전수기능을 수행하는 프로그람으로서 isPerfect( )메쏘드를 작성하여 주클라스의 main( )메쏘드에서 호출한다.

실례 3-5

```
Example 3-5 PerfectNum.java
1: public class PerfectNum
2: {
3:     public static void main(String args[])
4:     {
5:         int count = 1;
6:         for(int i = 1; i < 1000; i++)
7:         {
8:             if(isPerfect(i))
9:             {
10:                System.out.print(i + String.valueOf('\t'));
11:                count++;
12:            }
13:            if(count % 3 == 0)
14:            System.out.println( );
15:        }
16:    }
17:    static boolean isPerfect(int x)
18:    {
19:        int y = 0;
20:
21:        for(int i = 1; i < x; i++)
22:            if(x % i == 0)
23:                y += i;
24:        if(y == x)
25:            return true;
26:        else
27:            return false;
28:    }
29:}
```

프로그람설명

isPerfect( )메쏘드의 귀환값은 론리형이며 만일 실제파라메터에 들어가는 옹근수가 완전수이면 참값을 귀환하고 그렇지 않으면 거짓값을 귀환한다. 메쏘드의 장식부 역시 접근조종부와 비접근조종부로 나눈다. 여기에서는 비접근조종부를 가진 메쏘드들을 서술한다.

1) 추상메쏘드

장식부 abstract를 리용한 **추상메쏘드**는 메쏘드머리부만을 가질뿐 구체적인 메쏘드본체와 조작실현방법을 가지지 않는다. 실례로 아래의 전화거는 메쏘드 performDial()은 추상클라스 PhoneCard에서 정의한 추상메쏘드이다.

abstract void performDial();

보는바와 같이 abstract메쏘드는 메쏘드머리부의 선언만을 가지며 구분기호 《;》를 써서 메쏘드본체의 정의를 대신한다. 메쏘드를 정의하지 않는 리유를 그림 3-2를 통하여 보자. 여기서 전화카드종류는 모든 전화카드에서 추상해낸 공통적인 속성모임이며 매 전화카드는《전화걸기》의 기능을 가진다. 그러나 매 전화카드의《전화걸기》기능의 구체적인 실현 즉 구체적인 조작은 서로 같지 않다. 실례로 IC카드는 금액이 남아있기만 하면 IC카드전화기에 접속하여 통화할수 있다. 한편 200카드는 쌍음성주파수전화에서 우선 정확한 카드번호와 비밀번호를 입력해야 한다. 그러므로 PhoneCard의 서로 다른 하위클라스의 performDial( )메쏘드는 서로 같은 목적을 가질지라도 그 메쏘드본체는 서로 같지 않다.

이런 상황에서는 PhoneCard클라스에 메쏘드본체를 가지지 않는 추상메쏘드 performDial( )을 정의하고 메쏘드본체의 구체적인 실현은 현재클라스의 서로 다른 하위클라스들에서 그것들의 조작특성에 맞게 진행한다. 다시말하여 매 하위클라스는 상위클라스의 추상메쏘드를 계승한 후에 다시 서로 다른 명령문들로 메쏘드본체를 형성하여 그것을 새롭게 정의한다. 이때 이름과 귀환값 그리고 파라메터목록은 같지만 구체적인 조작은 차이나게 된다.

추상메쏘드를 사용하는 목적은 클라스의 모든 하위클라스들이 외적으로 서로 같은 이름의 메쏘드를 보여주어 하나의 통일적인 대면부를 만들자는데 있다.

사실상 abstract메쏘드에 대하여 메쏘드본체를 쓰기는 무의미하며 abstract메쏘드에 의거하는 abstract클라스는 자기의 객체를 가지지 않기때문에 오직 자기의 하위클라스를 가질 때에야 구체적인 객체가 존재하게 된다. 서로 다른 하위클라스는 이 abstract메쏘드에 대하여 서로 다른 실현메쏘드를 가지며 파라메터목록과 귀환값외의 다른 공통점은 없다. 그러므로 abstract메쏘드를 하나의 공통적인 대면으로 할수밖에 없다. 추상클라스 PhoneCard의 모든 하위클라스들에서는 이 대면을 사용하여 《전화걸기》의 기능을 완성한다.

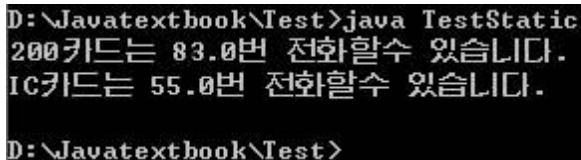
abstract메쏘드에 대한 정의 역시 일정한 우점을 가지고있다. 즉 구체적인 정보를 감출수 있으며 이 메쏘드를 호출하는 프로그람이 클라스와 하위클라스내부의 구체적인 상황에 너무 주목할 필요가 없다. 모든 하위클라스들이 사용하는것은 서로 같은 메쏘드머리부이고 메쏘드머리부안에는 실제상 이 메쏘드를 호출하는 프로그람명령문이 리해할수 있는 전체 정보를 포함하고있다. 그러므로 《전화걸기》조작을 완성해주는 명령문은 어느 전화카드의 하위클라스의 performDial( )메쏘드인가를 알 필요가 없으며 PhoneCard클라스의 performDial()메쏘드를 사용하기만 하면 된다.

여기서 주의해야 할것은 모든 추상메쏘드는 반드시 추상클라스안에 있어야 한다는것이다. 비추상클라스에서 추상메쏘드를 리용하여서는 안되며 추상클라스의 하위클라스가 추상클라스가 아닌 경우에는 반드시 상위클라스의 모든 추상메쏘드에 대하여 메쏘드본체를 작성해야 한다. 아래의 실례 3-6에서 전화카드종류를 추상클라스로 정의하고 IC카드와 200카드를 파생하여 performDial( )메쏘드에 대하여 메쏘드본체를 작성하였다.

실례 3-6

```
Example 3-6 TestAbstract.java
1: public class TestAbstract
2: {
3:     public static void main(String args[])
4:     {
5:         PhoneCard200 my200 = new PhoneCard200(50);
6:         IC_Card myIC = new IC_Card(50);
7:         System.out.println("200카드는 "+my200.TimeLeft()+"번 전화할수 있습니다.");
8:         System.out.println("IC카드는 "+myIC.TimeLeft()+"번 전화할수 있습니다.");
9:     }
10:}
11:abstract class PhoneCard
12:{
13:     double balance;
14:     abstract void performDial( );
15:     double TimeLeft( )
16:     {
17:         double current = balance;
18:         int times = 0;
19:         do
20:         {
21:             performDial( );
22:             times++;
23:         } while(balance> = 0);
24:         balance = current;
25:         return times-1;
26:     }
27:}
28:class PhoneCard200 extends PhoneCard
29:{
```

```
30:     static long nextCardNumber;
31:     static final String connectNumber = "200";
32:     static double additoryFee;
33:     long cardNumber;
34:     int password;
35:     boolean connected;
36:
37:     static
38:     {
39:         nextCardNumber = 2001800001;
40:         additoryFee = 0.1;
41:     }
42:     PhoneCard200(double ib)
43:     {
44:         cardNumber = nextCardNumber++;
45:         balance = ib;
46:     }
47:     void performDial( )
48:     {
49:         balance -= 0.5 + additoryFee;
50:     }
51:}
52: class IC_Card extends PhoneCard
53: {
54:     IC_Card(double ib)
55:     {
56:         balance = ib;
57:     }
58:      void performDial( )
59:     {
60:         balance -= 0.9;
61:     }
62:}
```

프로그람설명

실례 3-6에서는 4개의 클라스를 정의하고있다. 주클라스외에 추상클라스 PhoneCard와 2개의 하위클라스 PhoneCard200, IC_Card를 정의하고있다. PhoneCard는 balance마당과 2개의 메소드를 정의하고있으며 14행에서는 추상메소드 performDial( )을 정의하고있다. 15-26행은 전화카드의 남은 금액으로 아직 몇번

전화를 걸수 있는가를 검사하는 TimeLeft()메쏘드를 정의하고있다. 이 메쏘드는 우선 현재의 남은 금액을 보관하고 전화를 반복걸어 남은 금액이 령으로 될 때까지를 모의한 다음 남은 금액을 다시 회복하고 마지막에 아직 걸수 있는 전화회수를 귀환한다. 28-51행은 PhoneCard의 첫번째 하위클라스인 PhoneCard200을 정의하였다. 여기서 40행은 정적초기화기에서 부가비를 0.1로 설정하며 42-46행의 구성자는 새로운 전화카드금액을 지정하는 기능을 추가하였다. 47-50행은 상위클라스에서의 추상메쏘드 performDial( )을 계승하여 정의하였다. 200카드의 특징을 고려하여 0.5원의 통화비용과 부가비를 공제한다. 52-62행은 PhoneCard의 두번째 하위클라스인 IC_card를 정의하였다. 여기서 54-57행의 구성자는 새로운 IC카드의 금액을 지정한다.

58-61행은 상위클라스에서의 추상메쏘드를 IC_card판본의 performDial( )메쏘드로 다시 정의하였으며 그것은 매 통화에 대하여 balance로부터 0.9원의 비용을 공제한다. 3-9행의 main()메쏘드에서는 우선 PhoneCard클라스와 IC_Card클라스의 객체를 각각 창조한 다음 TimeLeft( )메쏘드를 호출하여 몇번 전화를 걸수 있는가를 시험검사한다. 21행에서 TimeLeft( )메쏘드는 performDial( )메쏘드를 호출하는데 그것이 어느 하위클라스의 performDial( )메쏘드인가에는 상관이 없다. 체계실행은 구체적인 객체에 따라 어느 클라스의 performDial( )메쏘드를 호출해야 하는가를 자동판단한다.

그림 3-5는 실례 3-6의 실행결과이다.

```
D:\Javatextbook\Test>java TestStatic
200카드는 83.0번 전화할수 있습니다.
IC카드는 55.0번 전화할수 있습니다.

D:\Javatextbook\Test>
```

그림 3-5. 실례 3-6의 실행결과

2) 정적메쏘드

static장식부를 리용한 메쏘드는 전체 클라스에 속하는 클라스메쏘드이다. 반면에 static장식부를 리용하지 않은 메쏘드는 어떤 구체적인 클라스객체나 실례에 속하는 메쏘드이다. 메쏘드를 static로 선언하려면 다음의 3가지를 고려하여야 한다.

· 메쏘드를 호출할 때 클라스이름을 사용하여 앞붙이를 만들어야 한다. 그러나 어떤 구체적인 객체이름은 안된다.

· static장식부를 리용하지 않은 메쏘드는 어떤 객체에 속하는 메쏘드이며 이 객체를 창조할 때 객체의 메쏘드는 기억기에서 자기전용의 코드부분을 가진다. 한편 static메쏘드는 전체 클라스에 속하는 메쏘드로서 그것은 기억기의 코드부분에서 클라스의 정의에 따라 분배적재하며 임의의 객체전용으로 되지 않는다.

· static메쏘드는 어떤 객체에 속하는 성원변수를 조종하고 처리할수 없으며 오직 전체 클라스에 속하는 성원변수만을 처리할수 있다. 즉 static메쏘드는 static마당

만을 처리할수 있다. 실례로 PhoneCard200에서 만일 부가비를 수정하려면 정적메쏘드 setAdditory( )를 정의해야 한다.

```
static void setAdditory(double newAdd)
{
    if(newAdd > 0)
        additoryFee = newAdd;
}
```

3) 최종메쏘드

final장식부를 리용하는 메쏘드로서 기능과 내부명령문이 더는 변경될수 없는 메쏘드이다. 즉 현재클라스의 하위클라스를 다시 정의할수 없는 메쏘드이다. 객체지향의 프로그람설계에서 하위클라스는 상위클라스로부터 계승한 어떤 메쏘드를 고쳐쓰고 다시 정의하며 상위클라스의 메쏘드와 같은 이름을 가질수 있다.

구체적인 실현과 기능들이 일치하지 않는 새로운 클라스의 메쏘드창조과정을 **다중정의**라고 한다. 만일 클라스의 어떤 메쏘드가 최종메쏘드라면 이 클라스의 하위클라스는 이 메쏘드이름과 같은 이름을 가진 자기의 메쏘드를 다시 새롭게 정의할수 없으며 상위클라스로부터 계승한 메쏘드만을 사용할수 있다. 이렇게 하면 이 메쏘드에 대응하는 구체적인 조작을 고정할수 있다. 주의할것은 이미 private장식부에 의하여 비공개로 한정된 모든 메쏘드(private장식부는 뒤에서 소개한다.)와 final클라스에 포함된 메쏘드들은 기정으로 final메쏘드로 인식한다. 이 메쏘드들이 하위클라스에서 계승될수 없든지 또는 근본적으로 하위클라스를 가지지 않아 다중정의할수 없으면 자연히 최종메쏘드로 된다.

4) 고유메쏘드

native장식부는 일반적으로 다른 언어를 리용하여 메쏘드본체를 작성하여 메쏘드기능을 구체적으로 실현하는 특수한 메쏘드를 선언하는데 쓰인다. 여기에서 다른 언어라고 할 때 C, C++, FORTRAN, 아쎔블러 등을 말한다. native메쏘드의 메쏘드본체는 다른 언어를 사용하여 프로그람의 밖에서 작성되기때문에 모든 native메쏘드는 메쏘드본체를 가지지 않으며 구분기호를 리용하여 바꾸어놓는다.

Java프로그람에서 다른 언어를 사용하여 작성한 모듈은 클라스메쏘드로 되는데 그 목적은 기본적으로 두가지이다. 즉 이미 존재하는 프로그람기능모듈을 충분히 리용하자는것과 반복작업을 피하자는것이다. Java는 해석형의 언어이므로 실행속도가 비교적 뜨며 그 어떤 최량화처리를 거치지 못하였을 때 Java프로그람의 실행속도는 C프로그람의 거의 15~20배정도이다. 실시간요구가 강하거나 실행효률를 높이는 경우 Java프로그람만을 사용하여 수요를 만족시킬수 없을 때 native메쏘드를 리용하여 실행속도가 높은 다른 언어의 방조를 받을수 있다.

Java프로그람에서 native를 사용할 때 특별히 주의하여야 할 점이 있다. native메쏘드가 다른 언어에 대응하여 작성한 모듈은 Java바이트코드가 아닌 2진코드형식으로써 Java프로그람에 삽입한것이고 이 2진코드는 보통 그것을 번역하는 기반에서

실행할수밖에 없으므로 모든 Java프로그람의 교차기반성능은 제한이 있다. 그러므로 이 메쏘드를 사용할 때는 특별히 심중하여야 한다.

5) 동기적메쏘드

synchronized로 장식한 메쏘드가 클라스의 메쏘드(즉 static의 메쏘드)이면 호출집행하기전에 체계클라스에서 현재 집행중의 클라스에 대응하는 객체를 잠근다.

synchronized로 장식한 메쏘드가 객체의 메쏘드(static를 사용하지 않고 장식한 메쏘드)이면 이 메쏘드를 호출집행하기전에 현재객체를 잠근다. synchronized장식부는 기본적으로 다중토막처리가 공존하는 프로그람에서의 조종과 동기화에 쓰인다.

제5절. 접근조종부

- **Java에서 클라스의 접근조종부는 오직 public만이다.**
- **접근조종부의 장식부: public, private, protected, private protected**
- **장식부는 한정된 범위에서 혼합하여 사용할수 있다.**

접근조종부는 클라스, 마당이나 메쏘드가 프로그람에 있는 다른 부분에 대하여 접근할수 있는가를 한정하는 장식부이다. 구체적으로 말하여 클라스와 그의 속성과 메쏘드에 대한 접근조종부는 프로그람에 있는 어느 다른 부분이 그것들에 접근할수 있고 어느 부분이 접근할수 없는가를 규정한다. 여기서 다른 부분이란 프로그람에서 이 클라스외의 다른 클라스를 가리키는것이며 장식부가 어떻게 정의되든지간에 클라스는 그 자체의 마당과 메쏘드에 접근하고 호출할수 있다. 그러나 이 클라스밖의 다른 부분이 이 마당이나 메쏘드에 접근할수 있는가 없는가는 이 마당과 메쏘드 및 그가 속하는 클라스의 접근조종부를 보아야 한다. 클라스의 접근조종부는 오직 public만이며 마당과 메쏘드의 접근조종부에는 4개 즉 public, private, protected, private protected가 있다. 그외에 접근조종부를 정의하지 않는 기정상황이 있다.

3.5.1.공개접근조종부

Java에서 클라스는 공개접근조종부인 public만을 가진다. 클라스를 공개클라스로 선언하면 모든 다른 클라스가 그것에 접근하고 인용할수 있다. 여기서 접근과 인용은 이 클라스를 완전히 볼수 있고 사용할수 있다는것을 말한다. 프로그람의 다른 부분은 이 클라스의 객체를 창조하고 이 클라스내부가 볼수 있는 성원변수와 그것을 호출하여 볼수 있는 메쏘드에 접근할수 있다.

Java의 클라스는 패키지의 개념으로부터 나온것이다. 패키지는 련관이 있는 클라스들의 모임이다. 같은 패키지에 있는 클라스들은 그 어떤 제한이 없이 편리하게

서로 접근 인용할수 있다. 한편 서로 다른 패키지의 클라스들사이에는 서로 볼수도 없고 인용할수도 없다. 그러나 클라스가 public로 선언되면 다른 패키지의 클라스에 접근할수 있다. 프로그람에서 import명령문을 사용하여 다른 패키지의 public클라스를 인입하는것에 의해 이 클라스에 접근하고 인용할수 있다.

public로 선언된 클라스는 완전히 공개적이지만 클라스안의 모든 마당과 메쏘드 역시 프로그람의 다른 부분에 보여질수 있다는것을 의미하지는 않는다. 클라스의 마당과 메쏘드에 접근할수 있는가를 보자면 이 마당과 메쏘드자체의 접근조종부를 보아야 한다. 만일 이 마당과 메쏘드자체의 접근조종부 역시 public이면 프로그람의 모든 다른 부분은 그것에 접근할수 있다.

클라스에서 public로 정의되는 메쏘드는 이 클라스의 대면부분이며 프로그람의 다른 부분은 그것들을 호출하여 현재의 클라스와 정보를 교환하고 통보문을 전달하며 나아가서 현재의 클라스에 영향을 주게 된다.

public를 리용하는 클라스의 마당을 공개마당이라고 한다. 만일 공개마당이 공개클라스에 속하면 그것은 모든 다른 클라스들에 의하여 인용될수 있다. public장식부는 안전성과 자료내장성을 저하시키므로 대체로 리용하지 말아야 한다.

3.5.2. 기정접근조종부

클라스가 접근조종부를 가지지 않으면 그것은 기정의 접근조종특성을 가지고있다는것을 나타낸다. 기정접근조종부는 클라스가 같은 패키지의 클라스에 의하여서만 접근, 인용될수 있고 다른 패키지의 클라스에 의하여서는 사용될수 없다는것을 의미한다. 이 접근특성을 **패키지접근성**이라고 한다. 클라스의 접근조종부선언을 통하여 전체 프로그람구조를 뚜렷하고 엄밀하게 할수 있으며 있을수 있는 클라스들사이의 간섭과 오유를 감소시킬수 있다.

또한 클라스안의 마당과 메쏘드가 접근조종부를 가지지 않으면 그것들은 패키지접근성을 가지고있다는것을 의미하며 같은 패키지안의 다른 클라스에 의하여 접근될수 있다.

3.5.3. 비공개접근조종부

private를 리용하여 장식하는 마당과 메쏘드는 클라스자체에 의하여서만 접근될수 있으며 임의의 다른 클라스(이 클라스의 하위클라스를 포함)에서 얻고 인용할수 없다. private장식부는 클라스의 비공개성원을 선언하는데 쓰이며 가장 높은 보호수준을 제공하고있다. 실례로 200전화카드클라스 PhoneCard200에서 전화카드의 비밀열쇠번호 password는 다른 클라스의 객체가 마음대로 호출리용하거나 수정하는것을 허용하지 않는다. 그러므로 이 마당은 비공개성원으로 선언할수 있다.

```
private int password;
```

다른 클라스가 비공개성원을 얻거나 수정하려면 클라스의 메쏘드로 실현하여야 한다. 실례로 PhoneCard클라스에서 getPassword( )메쏘드를 정의하여 비밀열쇠번호

를 얻을수 있다. setPassword( )를 정의하면 비밀번호를 수정할수 있으며 password를 완전히 포장하여 보호하기만 하면 클라스외부는 클라스내부가 비밀열쇠번호자료를 보존하고있다는것만을 알수 있으며 이 자료를 어느 변수에 보존하고 변수이름이 무엇인가는 알수 없다. 동시에 보증에 일정한 권한을 가지고있을 때에야 비밀열쇠번호를 찾아보거나 수정할수 있으며 getPassword( )메쏘드와 setPassword( )메쏘드에서 필요한 안전성검사를 하여 일정한 조건을 만족시킬 때에야 password의 수값을 얻거나 수정할수 있다.

3.5.4. 보호접근조종부

protected를 리용하여 장식하는 성원변수는 3개의 클라스(클라스자체, 같은 패키지의 다른 클라스, 다른 패키지에서 이 클라스의 하위클라스)에 의하여 인용될수 있다. protected장식부를 사용하는것은 다른 패키지에서 그것의 하위클라스를 리용하여 상위클라스의 속성에 접근하자는데 있다.

3.5.5. 비공개보호접근조종부

private와 protected를 결합하여 완전한 접근조종부를 구성한다.

즉 비공개보호접근조종부 private protected를 리용하여 장식하는 성원변수는 2가지 클라스에 의하여 접근, 인용될수 있다. 하나는 이 클라스자체이고 다른 하나는 이 클라스의 모든 하위클라스이다. 하위클라스는 이 클라스와 같은 패키지뿐만 아니라 다른 패키지에도 들어있다. private protected장식부는 같은 패키지내의 비하위클라스를 배제하여 성원변수가 명확한 계승관계를 가지도록 한다.

클라스, 속성과 메쏘드의 접근조종을 표 3-1과 그림 3-6에 보여주었다.

표 3-1. 클라스, 속성, 메쏘드의 접근조종

클라스 / 속성과 메쏘드	public	기정
public	A	B
protected	B+C	B
기정	B	B
private protected	C+D	E+D
private	D	D

그림 3-6에서 구역 A는 모든 클라스를 나타내고 구역 C는 현재클라스의 모든 하위클라스를 나타낸다. 그것들중에서 어떤것은 현재클라스와 같은 패키지에 있으며 어떤것은 현재클라스와 서로 다른 패키지에 있다. 구역 D는 현재클라스자체를 나타내

며 구역 E는 현재패키지의 현재클라스의 하위클라스를 나타낸다. 표 3-1은 클라스, 속성, 메소드의 접근조종을 보여준다.

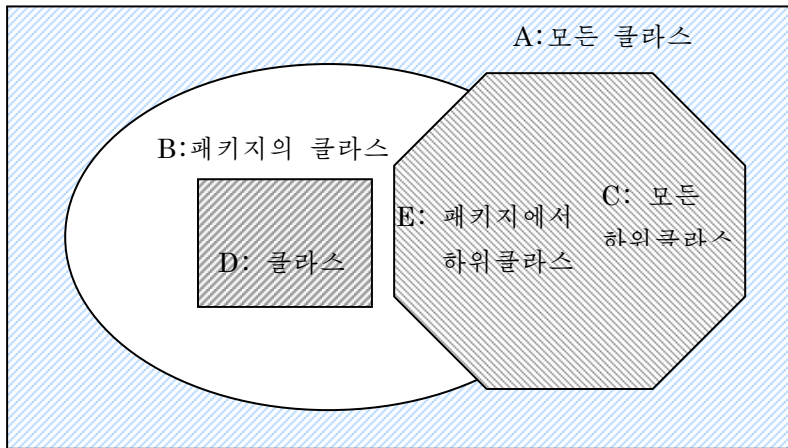



그림 3-6. 프로그람에서 접근조종구역

실례 3-7

```
Example 3-7 AccessControl.java
1: import java.applet.Applet;
2: import java.awt.*;
3:
4: public class AccessControl extends Applet //주클라스정의, 열람기에서 정보현시
5: {
6:     ClassBeAccessed c = new ClassBeAccessed( );
                                          //클라스에 접근하는 객체창조(구역 D)
7:     subClass sc = new subClass( ); //하위클라스에 접근하는 객체창조(구역 C)
8:     PackageClass ic = new PackageClass( );
                          //동일한 패키지의 클라스에 접근하는 객체창조(구역 B)
9:
10:    public void paint(Graphics g)  //접근가능한 정보를 현시
11:    {
12:        g.drawString("Self Accessible:", 10, 20);
                              //클라스는 자기의 모든 속성과 방법에 접근할수 있다.
13:        g.drawString(c.toString( ), 20, 35);
14:        g.drawString("Sub Accessible:", 10, 55);
                   //하위클라스는 상위클라스의 임의의 속성에도 직접 접근할수 있다.
15:        g.drawString(sc.AccessDirectly( ), 20, 70);
16:        g.drawString("Package Accessible:", 10, 90);
```

```
                //동일한 패키지의 클라스는 임의의 속성에도 접근할수 있다.
17:         g.drawString(ic.AccessDirectly( ), 20, 105);
18:         g.drawString("Access using public method:", 10, 125);
19:         g.drawString(sc.AccessCls( ), 20, 140);
20:         g.drawString(ic.AccessCls( ), 20, 155);
21:     }
22:}
23:class ClassBeAccessed //동일한 패키지의 다른 클라스는 이 객체를 창조할수 있다.
24:{
25:     public String m_PublicProperty; //공개속성
26:     String m_FriendlyProperty;       //기정속성
27:     protected String m_ProtectedProperty; //보호속성
28:     private String m_PrivateProperty;   //비공개속성
29:
30:     ClassBeAccessed( ) //구성자로서 매 속성에 초기값을 준다.
31:     {
32:         m_PublicProperty = new String("Public");
33:         m_FriendlyProperty = new String("Friendly");
34:         m_ProtectedProperty = new String("Protected");
35:         m_PrivateProperty = new String("Private");
36:     }
37:     public String toString( ) //공개메쏘드로서 매 속성들의 문자렬을 련결하고 현시
38:     {
39:         return(m_PublicProperty + ";"
40:             + m_FriendlyProperty + ";"
41:             + m_ProtectedProperty + ";"
42:             + m_PrivateProperty + ";");
43:     }
44:}
45:class subClass extends ClassBeAccessed   //클라스에 접근하는 부분클라스
46:{
47:     ClassBeAccessed c = new ClassBeAccessed( ); //클라스에 접근하는 객체창조
48:
49:     String AccessDirectly( ) //클라스에 접근하는 속성을 직접호출
50:     {
51:         return(c.m_PublicProperty + ";"   //공개속성
52:             + c.m_FriendlyProperty + ";" //기정속성
53:             + c.m_ProtectedProperty + ";"); //보호속성
54:     }
```

```
55:    String AccessCls( ) //클라스에 접근하는 공개메쏘드의 호출을 통하여
                                              그의 매 속성을 호출할수 있다.
56:    {
57:        return(c.toString( ));
58:    }
59:}
60:class PackageClass
61:{
62:    ClassBeAccessed c = new ClassBeAccessed( ); //클라스에 접근하는 객체창조
63:
64:    String AccessDirectly( ) //클라스에 접근하는 속성의 직접호출
65:    {
66:        return(c.m_PublicProperty + ";"
67:            + c.m_FriendlyProperty + ";"
             + c.m_ProtectedProperty + ";");
68:    }
69:    String AccessCls( )
70:    {
71:        return(c.toString( )); //클라스에 접근하는 공개메쏘드의 호출을
                                         통하여 그의 매 속성을 호출할수 있다.
72:    }
73:}
```

프로그람설명

실례 3-7프로그람의 실행결과를 그림 3-7에서 보여준다.

그림 3-7로부터 접근조종부의 접근조종권한에 대한 제한작용을 똑똑히 볼수 있다. 많은 경우에 장식부는 혼합하여 사용한다. 클라스의 3개 장식부 public, final과 abstract사이에는 서로 배척하지 않으며 공개클라스는 추상일수 있다.

public abstract class transportmeans...

공개클라스는 final일수도 있다. 실례로 public final class Socket…

장식부를 혼합하여 리용할 때 주의할 점:

· final과 abstract는 동시에 사용할수 없다.

· abstract는 private, static, final 또는 native와 나란히 같은 메쏘드를 장식할수 없다.

· abstract클라스에서 private의 성원을 가질수 없다.(속성과 메쏘드 포함)

· abstract메쏘드는 반드시 abstract클라스에 있어야 한다.

· static메쏘드에서 static가 아닌 속성을 처리할수 없다.

그림 3-7. 실례 3-7의 실행결과

제4장. 계승과 다형성

이 장에서는 대상지향프로그람설계의 중요한 특징인 계승과 다형성에 대하여 서술한다. 계승은 대상지향프로그람설계방법에서 중요한 수단이다. 계승을 통하여 프로그람구조를 보다 효과적으로 구성할수 있으며 클라스사이의 관계를 명확히 하고 이미 있는 클라스들을 충분히 리용하여 보다 복잡하고 심도가 깊은 개발을 완성할수 있다.

다형성은 클라스의 추상도와 내장성을 높일수 있게 하고 하나이상의 련관클라스에 대한 대면을 통일시킬수 있다. 이 장의 뒤부분에서 대면부와 패키지에 대하여 서술한다.

제1절. 계승

- **계승은 상위클라스와 하위클라스와의 관계이다.**
- **Java언어는 단일계승만을 지원한다.**
- **Java에서의 계승은 extends예약어를 리용하여 실현한다.**
- **this와 super는 현재객체와 상위객체를 대신하는 예약어이다.**

4.1.1. 계승의 개념

객체지향기술의 특징중에서 계승은 제일 중요하다. **계승**은 실제상 객체지향프로그람에서 두 클라스사이에 존재하는 하나의 관계이다. 하나의 클라스가 다른 클라스의 모든 자료와 조작들을 자기의 부분이나 성원으로 할 때 이 두 클라스사이에는 계승관계가 있다고 말한다. 계승시키는 클라스를 **상위클라스** 혹은 **초클라스**라고 하며 상위클라스의 자료와 조작들을 계승하는 모든 클라스를 **하위클라스**라고 한다. 상위클라스는 동시에 여러개의 하위클라스를 가질수 있는데 이때 상위클라스는 모든 하위클라스들의 공통적인 마당과 공통적인 메쏘드들의 모임이며 매개 하위클라스는 상위클라스의 특수화로서 마당과 메쏘드에 대하여 기능, 속성에서의 확장과 연장이다.

전화카드를 실례로 고찰하자. 그림 4-1은 매 전화카드클라스의 계층적구조, 마당, 메쏘드를 렬거하고있다. 그림 4-1에서 볼수 있는것처럼 객체지향의 이 계승관계는 실제상 인간의 일상사유방식에 부합된다. 전화카드는 전화번호가 없는것과 있는것으로 나누어진다. 번호가 없는 전화카드는 자기카드, IC카드 등으로 세분되며 번호가 있는 전화카드는 IP전화카드와 200전화카드 등으로 나눌수 있다. 여기서 전화카드클라스는 모든 클라스들의 상위클라스이며 모든 전화카드의 공통적인 속성모임이다. 이 공통적인 속성에는 카드에 남은 금액 등의 정적인 자료속성과 전화걸기, 남은 금액의 조사 등의 동적인 행위속성도 포함된다.

전화카드를 구체화하면 2개의 하위클라스 즉 전화번호가 없는 전화카드와 전화번호가 있는 전화카드로 각각 분류된다. 여기서 하나는 상위클라스인 전화카드의 모든

속성(마당과 메쏘드를 포함)을 계승하여 남은 금액, 전화걸기, 남은 금액의 조사 등의 자료와 조작을 포함하며 다른 하나는 상위클라스의 속성외에 자기의 고유한 특수속성(례: 번호가 있는 모든 전화카드에 대하여서는 카드번호, 비밀번호, 접수번호 등의 마당과 교환기를 등록하는 행위를 가져야 한다.)들을 정의하고있다. 이 속성들은 번호가 없는 전화카드에 대하여서는 없다. 번호가 있는 전화카드로부터 IP전화카드와 200전화카드까지의 계승은 완전히 서로 같은 원칙을 준수하고있다.

계승을 사용하는 기본우점은 프로그람구조를 구체화하고 코드작성과 유지의 작업량을 낮춘다는것이다. 그림 4-1에서 보면 남은 금액은 모든 전화카드가 공유하는 속성이다. 첫째 실현방안은 매 전화카드클라스에 대하여 자기의 남은 금액마당을 정의하는것이다. 둘째 실현방안은 추상적인 전화카드상위클라스에서 남은 금액마당을 정의하는것이다. 다른 클라스들은 그것에 의해 계승된다. 첫째 방안은 둘째 방안에 비하여 코드량은 몇배 더 많게 된다. 또한 공통속성에 수정할것이 생길 때 첫째 방안은 매 클라스에서 상응한 수정을 하여야 하지만 둘째 방안은 상위클라스에서 한번만 수정하면 되므로 유지작업량이 크게 감소될뿐아니라 첫째 방안에서 생길수 있는 수정오유를 피할수 있다.

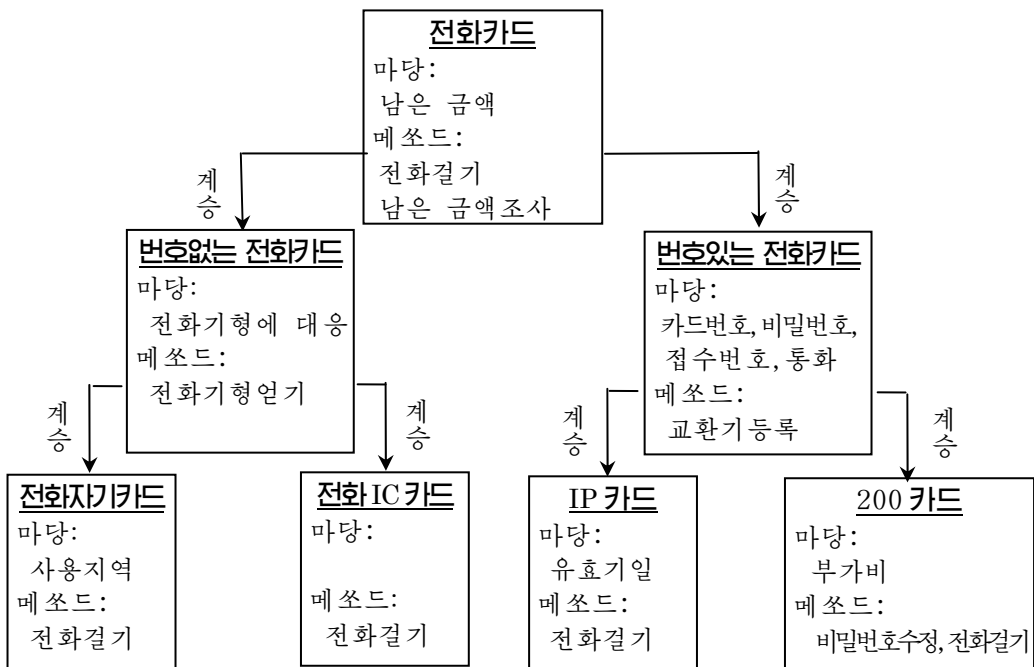



그림 4-1. 전화카드클라스와 그것들사이의 계승관계

객체지향의 계승성에는 단일계승과 다중계승에 대한 개념이 있다. **단일계승**은 임의의 클라스가 오직 하나의 상위클라스만을 가진다는것을 의미하며 **다중계승**은 클라

스가 하나이상의 상위클라스를 가질수 있다는것을 의미한다. 다중계승에서 정적인 자료속성과 조작들은 그의 모든 상위클라스로부터 계승된다.

단일계승을 리용한 프로그람구조는 비교적 간단하며 그림 4-1에서 보여주는것처럼 단순한 나무구조이고 조종하기가 상대적으로 쉽다.

반대로 다중계승을 지원하는 프로그람은 구조가 복잡한 그물형태이고 설계와 실현이 비교적 복잡하다. 그러나 현실세계의 실제문제를 해결하는데서 대상문제들이 내부구조가 더 복잡한 그물구조로 되여있으므로 다중계승의 프로그람을 리용하여 모의하기가 비교적 자연스럽다. 만일 단일계승프로그람으로 이 문제를 해결하려면 다른 보조적인 조치를 취해야 한다. C++는 개발자들이 익숙한 다중계승을 지원하는 객체지향의 프로그람작성언어이지만 이 책에서 소개하는 Java언어는 안전, 믿음성을 고려하여 나왔으므로 단일계승만을 지원한다.

4.1.2. Java에서의 계승

1) 하위클라스의 파생

Java에서의 계승은 extends예약어를 리용하여 실현한다. 클라스를 정의할 때 extends예약어를 사용하여 새롭게 정의하는 클라스의 상위클라스를 지정하므로써 두 클라스사이에 계승관계를 명백히 할수 있다. 새롭게 정의하는 클라스를 하위클라스라고 하며 상위클라스로부터 모든 비공개인 속성과 메쏘드를 계승하여 자기의 성원으로 한다.

실례 4-1

```
Example 4-1 PhoneCard.java //그림 4-1의 전화카드류형의 계승구조를 실현
1: import java.util.*;
2: abstract class PhoneCard
3: {
4:      double balance;
5:
6:      abstract boolean performDial( );
7:      double getBalance( )
8:      {
9:          return balance;
10:     }
11:}
12:abstract class None_Number_PhoneCard extends PhoneCard
13:{
14:     String phoneSetType;
15:
```

```
16:     String getSetType( )
17:     {
18:         return phoneSetType;
19:     }
20:}
21:abstract class Number_PhoneCard extends PhoneCard
22:{
23:     long cardNumber;
24:     int password;
25:     String connectNumber;
26:     boolean connected;
27:
28:     boolean performConnection(long cn, int pw)
29:     {
30:         if(cn == cardNumber && pw == password)
31:         {
32:             connected = true;
33:             return true;
34:         }
35:         else
36:             return false;
37:     }
38:}
39:class magCard extends None_Number_PhoneCard
40:{
41:     String usefulArea;
42:
43:     boolean performDial( )
44:     {
45:         if(balance > 0.9)
46:         {
47:             balance -= 0.9;
48:             return true;
49:         }
50:         else
51:             return false;
52:     }
53:}
54:class IC_Card extends None_Number_PhoneCard
55:{
```

```
56:     boolean performDial( )
57:     {
58:         if(balance > 0.5)
59:         {
60:             balance -= 0.9;
61:             return true;
62:         }
63:         else
64:             return false;
65:     }
66:}
67:class IP_Card extends Number_PhoneCard
68:{
69:     Date expireDate;
70:     boolean performDial( )
71:     {
72:         if(balance > 0.3 && expireDate.after(new Date( )))
73:         {
74:             balance -= 0.3;
75:             return true;
76:         }
77:         else
78:             return false;
79:     }
80:}
81:class D200_Card extends Number_PhoneCard
82:{
83:     double additoryFee;
84:
85:     boolean performDial( )
86:     {
87:         if(balance > (0.5 + additoryFee))
88:         {
89:             balance -= (0.5 + additoryFee);
90:             return true;
91:         }
92:         else
93:             return false;
94:     }
95:}
```

프로그람설명

실례 4-1은 PhoneCard, None_Number_PhoneCard, Number_ PhoneCard, magCard, IC_Card, IP_Card, D200_Card의 7개 클라스를 정의하고있다. 여기에서 None_Number_PhoneCard, Number_PhoneCard클라스는 PhoneCard클라스가 파생한 하위클라스이며 magCard, IC_Card클라스는 None_Number_PhoneCard클라스가 파생한 하위클라스이다. 그리고 IP_Card, D200_Card클라스는 Number_PhoneCard 클라스가 파생한 하위클라스이다.

실례 4-1의 프로그람에서는 4행(PhoneCard클라스)에서만 마당 balance를 정의하고있다. 그러나 45, 47(magCard클라스), 58, 60(IC_Card클라스), 72, 74(IP_Card클라스), 87, 89(D200_Card클라스)행에서 balance마당을 사용하고있는데 모두 상위클라스 PhoneCard로부터 계승한것이다. 이밖에 PhoneCard클라스는 6행에서 추상메쏘드 performDial( )을 정의하고있다. 또한 파생된 4개의 전화카드클라스는 추상클라스가 아니며 자기의 구체적정황에 맞게 performDial( )메쏘드를 각각 정의하고있다.

69행은 java.util패키지의 체계클라스 Date를 사용하는데 매개 Date클라스의 객체는 구체적인 날자를 의미한다. 72행의 new Date( )는 현재날자를 포함하는 Date클라스의 객체를 창조한다. after( )메쏘드는 Date클라스의 메쏘드이며 실효날자가 현재날자보다 늦을 때 expireDate.after(new Date( ))는 true를 귀환하고 그렇지 않으면 false를 귀환한다.

2) 마당의 계승과 감추기

(1) 마당의 계승

하위클라스는 상위클라스의 모든 공개마당을 계승할수 있다. 실례로 매 전화카드클라스가 포함하는 마당은 각각 다음과 같다.

```
PhoneCard클라스:
        double balance;
None_Number_PhoneCard클라스:
        double balance; // 상위클라스 PhoneCard로부터 계승
        String phoneSetType;
Number_PhoneCard클라스:
        double balance; // 상위클라스 PhoneCard로부터 계승
        long cardNumber;
        int password;
        String connectNumber;
        boolean connect;
magCard클라스:
        double balance; // 상위클라스 None_Number_PhoneCard로부터 계승
        String phoneSetType; // 상위클라스 None_Number_PhoneCard로부터 계승
        String usefulArea;
```

IC_Card클라스:

double balance; // 상위클라스 None_Number_PhoneCard로부터 계승

String phoneSetType; // 상위클라스 None_Number_PhoneCard로부터 계승

IP_Card클라스:

double balance; // 상위클라스 Number_PhoneCard로부터 계승

long cardNumber; // 상위클라스 Number_PhoneCard로부터 계승

int password; // 상위클라스 Number_PhoneCard로부터 계승

String connectNumber; // 상위클라스 Number_PhoneCard로부터 계승

boolean connect; //상위클라스 Number_PhoneCard로부터 계승

Date expireDate;

D200_Card클라스:

double balance; // 상위클라스 Number_PhoneCard로부터 계승

long cardNumber;// 상위클라스 Number_PhoneCard로부터 계승

int password; // 상위클라스 Number_PhoneCard로부터 계승

String connectNumber; // 상위클라스 Number_PhoneCard로부터 계승

boolean connect; // 상위클라스 Number_PhoneCard로부터 계승

double additoryFee;

보는바와 같이 상위클라스의 모든 공개마당은 실제상 매 하위클라스가 가지고있는 마당으로 된다. 하위클라스가 상위클라스로부터 마당을 계승한다는것은 상위클라스마당의 정의부분을 복사한다는것을 의미하지 않는다.

(2) 마당의 은페

하위클라스는 상위클라스로부터 계승한 마당변수와 완전히 같은 변수를 다시 정의할수 있는데 이것을 **마당의 은페**라고 한다. 실례 4-1의 81-95행에서 정의한 D200_Card메쏘드를 아래와 같이 수정할수 있다.

```
82:class D200_Card extends Number_PhoneCard
81:{
83:    double additoryFee;
84:    double balance;
85:    boolean performDial( )
86:    {
87:        if(balance > (0.5 + additoryFee))
88:        {
89:            balance -= (0.5 + additoryFee);
90:            return true;
91:        }
```

```
92:        else
93:            return false;
94:    }
95:}
```

84행에서는 상위클라스로부터 계승한 balance변수와 완전히 같은 변수를 추가하여 정의하였다. 이렇게 수정하면 D200_Card클라스의 마당은 아래와 같이 변한다.

D200_Card클라스:

double balance; //상위클라스 Number_PhoneCard로부터 계승

double balance; //D200_Card클라스가 자체로 정의한 마당

long cardNumber; // 상위클라스 Number_PhoneCard로부터 계승

int password; // 상위클라스 Number_PhoneCard로부터 계승

String connectNumber; // 상위클라스 Number_PhoneCard로부터 계승

boolean connect; //상위클라스 Number_PhoneCard로부터 계승

double additoryFee;

하위클라스에서는 상위클라스와 같은 이름의 속성변수를 정의하였다. 즉 상위클라스변수에 대한 은페를 진행하였다. 여기서 은페라는것은 하위클라스가 두개의 같은 이름을 가진 변수를 정의하는것을 말한다. 하나는 상위클라스로부터 계승한 변수이며 다른 하나는 하위클라스자체가 정의한 변수이다. 이때 상위클라스로부터 계승한 변수를 《은페》한다고 말한다.(실례 4-2)

실례 4-2

```
Example 4-2 TestHiddenField.java
1: public class TestHiddenField
2: {
3:     public static void main(String args[])
4:     {
5:         D200_Card my200 = new D200_Card( );
6:         my200.balance = 50;
7:         System.out.println("상위클라스의 은페에 의한 금액:"+my200.getBalance());
8:         if(my200.performDial( ))
9:         System.out.println("하위클라스의 남은 금액:" + my200.balance);
10:    }
11:}
12:abstract class PhoneCard
13:{
14:    double balance;
```

```
15:
16:     abstract boolean performDial( );
17:     double getBalance( )
18:     {
19:         return balance;
20:     }
21:}
22:abstract class Number_PhoneCard extends PhoneCard
23:{
24:     long cardNumber;
25:     int password;
26:     String connectNumber;
27:     boolean connected;
28:
29:     boolean performConnection(long cn, int pw)
30:     {
31:         if(cn == cardNumber && pw == password)
32:         {
33:             connected = true;
34:             return true;
35:         }
36:         else
37:             return false;
38:     }
39:}
40:class D200_Card extends Number_PhoneCard
41:{
42:     double additoryFee;
43:     double balance;
44:
45:     boolean performDial( )
46:     {
47:         if(balance > (0.5 + additoryFee))
48:         {
49:             balance -= (0.5 + additoryFee);
50:             return true;
51:         }
52:         else
53:             return false;
54:     }
55:}
```

프로그람설명

실례 4-2에서 5행은 D200_Card클라스의 객체 my200을 창조한다. 이 객체는 2개의 balance변수를 가지는데 하나는 상위클라스 phoneCard로부터 계승된것이고 다른 하나는 43행에서 다시 정의한 자체의 balance변수이다. 6행은 마당은페원칙에 따라 my200자체의 balance변수에 값주기한다. 7행은 my200객체의 getBalance( )메쏘드의 귀환값을 출력한다. 여기서 getBalance( )메쏘드는 상위클라스 PhoneCard에서 정의한것이며 그것의 귀환값은 my200객체가 상위클라스PhoneCard로부터 계승한 balance변수의 수값이다. 이 balance변수는 값주기하지 않았으므로 기정으로 0이다. 8행은 자체의 balance변수를 수정한다. 9행은 전화를 건 다음 my200객체의 balance변수의 값을 출력한다. 그림 4-2는 실례 4-2의 실행결과이다.

```
D:\Javatextbook\Test>javac TestHiddenField.java

D:\Javatextbook\Test>java TestHiddenField
상위클라스의 은페에 의한 금액:0.0
하위클라스의 남은 금액:49.5

D:\Javatextbook\Test>
```

그림 4-2. 실례 4-2의 실행결과

3) 메쏘드의 계승과 다중정의

(1) 메쏘드의 계승

상위클라스의 공개메쏘드는 하위클라스에 계승될수 있다. 실례 4-2의 7행에서 정의한 my200객체의 getBalance( )메쏘드는 바로 상위클라스 PhoneCard에서 계승된것이다. 메쏘드의 계승관계에 따라 매 전화카드가 포함하는 메쏘드(메쏘드머리부만을 현시)는 아래와 같다.

PhoneCard클라스:

```
abstract boolean performDial( )
double getBalance( )
```

None_Number_PhoneCard클라스:

```
abstract boolean performDial( ) // 상위클라스 PhoneCard로부터 계승
double getBalance( ) // 상위클라스 PhoneCard로부터 계승
String getSetType( )
```

Number_PhoneCard클라스:

```
abstract boolean performDial( ) // 상위클라스 PhoneCard로부터 계승
double getBalance( ) // 상위클라스 PhoneCard로부터 계승
boolean performConnection(long cn,int pw)
```

magCard클라스:

double getBalance() // 상위클라스 None_Number_PhoneCard로부터 계승

String getSetType() // 상위클라스 None_Number_PhoneCard로부터 계승

boolean performDial()

IC_Card클라스:

double getBalance() // 상위클라스 None_Number_PhoneCard로부터 계승

String getSetType() // 상위클라스 None_Number_PhoneCard로부터 계승

boolean performDial()

IP_Card클라스:

boolean performDial()

double getBalance() // 상위클라스 Number_PhoneCard로부터 계승

boolean performConnection(long cn,int pw)

// 상위클라스 Number_PhoneCard로부터 계승

D200_Card클라스:

boolean performDial()

double getBalance() // 상위클라스 Number_PhoneCard로부터 계승

boolean performConnection(long cn,int pw)

// 상위클라스 Number_PhoneCard로부터 계승

이와 같이 매 클라스의 객체는 상위클라스로부터 계승한 메쏘드를 자유롭게 사용할수 있다.

(2) 메쏘드의 다중정의

하위클라스가 상위클라스와 같은 이름의 마당을 정의할수 있는것처럼 역시 상위클라스와 같은 이름의 메쏘드를 다시 정의할수 있다. 이것을 상위클라스에 대하여 **메쏘드의 다중정의**(Overload)를 실현한다고 말한다.

메쏘드의 다중정의는 마당의 은폐와는 같지 않다. 하위클라스에서 상위클라스의 은폐된 마당을 볼수 없을 뿐이지 둘다 자기의 독립적인 기억공간을 차지한다. 그러나 메쏘드다중정의에서는 상위클라스메쏘드가 차지한 기억기를 없애버리며 이로부터 상위클라스메쏘드가 하위클라스객체에서 중복없이 존재하게 된다. 실례 4-1에서 매 전화카드클라스인 magCard, IC_Card, IP_Card, D200_Card는 자기의 performDial() 메쏘드를 정의하고있으며 따라서 그것들이 상위클라스로부터 계승한 추상적인 performDial()은 존재하지 않는다. 실례 4-3에서는 실례 4-2의 D200_Card클라스에서 상위클라스에서 정의한 getBalance()메쏘드를 다중정의하였다. 실행결과 getBalance()메쏘드를 리용하여 my200객체자체의 balance마당을 귀환시킨다.

실례 4-3

```
Example 4-3 TestOverLoad.java
1: public class TestOverLoad
2: {
3:      public static void main(String args[])
4:      {
5:          D200_Card my200 = new D200_Card( );
6:          my200.balance = 50;
7:          System.out.println("상위클라스의 은페에 의한 금액:"+my200.getBalance());
8:          if(my200.performDial( ))
9:          System.out.println("하위클라스의 남은 금액:"+my200.balance);
10:     }
11:}
12:abstract class PhoneCard
13:{
14:     double balance;
15:
16:     abstract boolean performDial( );
17:     double getBalance( )
18:     {
19:         return balance;
20:     }
21:}
22:abstract class Number_PhoneCard extends PhoneCard
23:{
24:     long cardNumber;
25:     int password;
26:     String connectNumber;
27:     boolean connected;
28:
29:     boolean performConnection(long cn, int pw)
30:     {
31:         if(cn == cardNumber && pw == password)
32:         {
33:             connected = true;
34:             return true;
35:         }
36:         else
37:             return false;
38:     }
39:}
```

```
40:class D200_Card extends Number_PhoneCard
41:{
42:     double additoryFee;
43:     double balance;
44:
45:     boolean performDial( )
46:     {
47:          if(balance > (0.5 + additoryFee))
48:          {
49:               balance   -= (0.5 + additoryFee);
50:               return true;
51:          }
52:          else
53:               return false;
54:     }
55:     double getBalance( )
56:     {
57:           return balance;
58:     }
59:}
```

```
D:₩Javatextbook₩Test>javac TestOverLoad.java

D:₩Javatextbook₩Test>java TestOverLoad
상위클라스의 은페에 의한 금액:50.0
하위클라스의 남은 금액:49.5

D:₩Javatextbook₩Test>
```

그림 4-3. 실례 4-3의 실행결과

메쏘드의 다중정의에서 주의하여야 할 문제는 하위클라스에서 상위클라스가 이미 가지고있는 메쏘드를 다시 정의할 때 상위클라스와 완전히 같은 메쏘드머리부선언을 진행해야 한다는것이다. 즉 상위클라스와 완전히 같은 메쏘드이름과 귀환값, 파라메터목록을 가져야 한다.

4) this와 super

this와 super는 현재객체와 상위클라스의 객체를 대신하는 예약어이다. Java체계에서 매개 클라스는 암시적으로 null, this, super의 3개 마당으로 표현할수 있다. 여기서 null은 빈값을 의미한다. 객체를 정의하였으나 이에 대해 기억기를 확보하지 않았을 때 이 객체를 null로 지정할수 있다. this와 super마당은 계승과 밀접한 관계를 가진다.

(1) this

this가 표시하는것은 현재객체자체이며 보다 정확히 말하면 현재객체에 대한 인용을 의미한다. 객체의 인용은 객체의 또 다른 이름으로 리해할수 있으며 인용을 통하여 객체에 접근할수 있다. 하나의 객체는 몇개의 인용을 가질수 있는데 this가 바로 그중의 하나이다. this를 리용하면 현재객체의 메쏘드를 호출하거나 현재객체의 마당을 사용할수 있다. 실례로 D200_Card클라스에서의 getBalance( )메쏘드는 같은 객체의 마당 balance를 호출하므로 this를 리용하여 아래와 같이 실현할수 있다.

```
double getBalance( )
{
    return this.balance;
}
```

우에서 귀환하는것은 현재의 같은 객체의 balance마당이다. 이 경우에 this를 추가하지 않을수도 있다. 보다 복잡한 정황에서 this는 현재객체의 인용을 파라메터로 하여 다른 객체나 메쏘드에 전달한다. 실례로 도형사용자대면부의 Java Applet프로그람(실례 2-3)을 고찰하자.

```
Example 2-3 getDouble.java
1: import java.applet. * ;
2: import java.awt. * ;
3: import java.awt.event. * ;
4:
5: public class getDouble extends Applet implements ActionListener
6: {
7:     Label prompt;
8:     TextField input;
9:     double d = 0.0;
10:
11:    public void init( )
12:    {
13:        prompt = new Label("류동소수점수를 입력하십시오:");
14:        input = new TextField(10);
15:        add(prompt);
16:        add(input);
17:        input.addActionListener(this);
18:    }
19:    public void paint(Graphics g)
20:    {
21:        g.drawString("입력한 자료:" + d, 10, 50);
22:    }
```

```
23:    public void actionPerformed(ActionEvent e)
24:    {
25:        d = Double.valueOf(input.getText( )).doubleValue( );
26:        repaint( );
27:    }
28:}
```

프로그람설명

실례에서 17행의 addActionListener( )메쏘드는 체계클라스 TextField의 메쏘드이며 이 메쏘드를 호출하여 ActionListener대면을 실현한 객체를 실제파라메터로 제공할수 있다. 5행에서 정의한 사용자클라스 getDouble은 implements예약어(대면 및 그의 실현은 이 장의 마지막에서 서술한다.)를 리용하여 ActionListener대면을 실현하고있으며 this를 사용하여 현재 getDouble클라스의 객체를 addActionListener( )메쏘드를 호출하는 실제파라메터로 지정한다.

(2) super

super로 표시되는것은 객체의 직접적인 상위클라스객체이며 보다 정확히 말하면 객체의 직접적인 상위클라스객체의 인용이다. 직접적인 상위클라스라는것은 현재객체와 직접적인 계승관계가 있는 조상클라스를 말한다. 례를 들어 클라스 A가 하위클라스 B를 파생하고 B클라스는 또 자기의 하위클라스 C를 파생한다고 가정하면 B는 C의 직접적인 상위클라스이고 A는 C의 조상클라스이다. 실례 4-3에서 Number_phoneCard클라스는 D200_Card클라스의 직접적인 상위클라스이며 PhoneCard클라스는 D200_Card클라스의 조상클라스이다. Super로 표시하는것은 바로 직접적인 상위클라스이다.

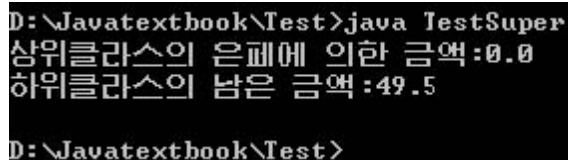
실례 4-4

```
Example 4-4 TestSuper.java
1: public class TestSuper
2: {
3:     public static void main(String args[])
4:     {
5:         D200_Card my200 = new D200_Card( );
6:         my200.balance = 50;
7:         System.out.println("상위클라스의 은페에 의한 금액:" + my200.getBalance( ));
8:         if(my200.performDial( ))
9:             System.out.println("하위클라스의 남은 금액:" + my200.balance);
10:     }
11:}
```

```
12:abstract class PhoneCard
13:{
14:     double balance;
15:
16:     abstract boolean performDial( );
17:     double getBalance( )
18:     {
19:         return balance;
20:     }
21:}
22:abstract class Number_PhoneCard extends PhoneCard
23:{
24:     long cardNumber;
25:     int password;
26:     String connectNumber;
27:     boolean connected;
28:
29:     boolean performConnection(long cn, int pw)
30:     {
31:         if(cn == cardNumber && pw == password)
32:         {
33:             connected = true;
34:             return true;
35:         }
36:         else
37:             return false;
38:     }
39:}
40:class D200_Card extends Number_PhoneCard
41:{
42:     double additoryFee;
43:     double balance;
44:
45:     boolean performDial( )
46:     {
47:         if(balance > (0.5 + additoryFee))
48:         {
49:             balance   -= (0.5 + additoryFee);
50:             return true;
```

```
51:         }
52:         else
53:             return false;
54:     }
55:     double getBalance( )
56:     {
57:         return super.balance;
58:     }
59:}
```

프로그람설명

실례 4-4에서 57행의 getBalance( )메쏘드가 귀환시키는것은 현재객체에 대한 super마당의 balance변수이다. 현재객체의 super마당은 D200_Card클라스의 직접적인 상위클라스 Number_phoneCard의 인용이고 Number_PhoneCard의 balance변수는 PhoneCard클라스로부터 계승한것이다. 그러므로 실례 4-4에서 getBalance( )메쏘드가 귀환시키는것은 여전히 값주기가 없는 상위클라스의 balance변수이다.

여기서 주의해야 할것은 this와 super는 클라스에 속하는 특별히 지적한 마당으로서 현재객체와 객체의 상위객체를 대표할 뿐 다른 클라스의 속성과 같이 마음대로 인용할수 없다는것이다. 아래의 명령문의 사용법은 틀린것이다.

```
D200_Card my200 = new D200_Card( );
my200.this.getBalance( );     //오유
my200.super.getBalance( );    //오유
```

그림 4-4는 실례 4-4의 실행결과이다.

```
D:\Javatextbook\Test>java TestSuper
상위클라스의 은페에 의한 금액:0.0
하위클라스의 남은 금액:49.5

D:\Javatextbook\Test>
```

그림 4-4. 실례 4-5의 실행결과

제2절. 다형성

- 다형은 같은 이름의 서로 다른 메쏘드가 공존하는것이다.
- 다형성은 계승관계에서만 성립한다.
- Person person = new Student( )

다형성은 객체지향프로그람설계에서 또 하나의 중요한 특성이다. 수속지향의 언어를 리용하여 프로그람을 작성하는데서 중요한것은 수속이나 함수를 작성하는것이다. 이 수속과 함수는 각자 일정한 기능에 대응하며 그 이름이 중복되여서는 안된다. 그렇지 않으면 호출할 때 례외와 오유를 발생시킬수 있다. 그러나 객체지향프로그람설계에서는 이러한 《중복》을 리용하여 프로그람의 추상도와 간결성을 높여야 하는 경우가 있다.

그림 4-1의 전화카드나무구조를 고찰해보면 모든 전화카드에 대하여 《전화걸기》조작의 구체적실현이 같지 않다. 실례로 자기카드의 《전화걸기》는 《자기카드전화기를 찾아 직접 전화번호를 돌리기》이며 200카드의 《전화걸기》는 《쌍음성주파수전화기를 찾고 우선 카드번호, 비밀번호를 입력한 후에 번호를 돌린다.》이다. 그러나 이 목표와 최종기능이 서로 같은 프로그람이 다른 이름을 사용한다면 반드시《자기전화걸기》, 《200카드전화걸기》 등 많은 메쏘드를 각각 정의하여야 한다. 이렇게 하면 계승이라는 우월성이 나타나지 않는다.

객체지향의 프로그람설계에서는 이 문제를 해결하기 위하여 다형의 개념을 도입하였다. **다형**이라는것은 프로그람에서 같은 이름의 서로 다른 메쏘드가 공존하는것을 말한다. 객체지향의 프로그람에서 다형의 경우는 여러가지이다. 하위클라스에서 상위클라스메쏘드를 다중정의하여 다형을 실현할수도 있고 동일한 클라스에서 같은 이름으로 정의하여 서로 다른 메쏘드로 리용할수도 있다.

1) 다형의 첫째 류형

전화카드를 실례로 고찰하자. PhoneCard클라스는 매 하위클라스들이 공유하고있는 메쏘드인 《전화걸기》를 가지고 전화걸기기능을 나타낸다. 계승의 특징에 따라 PhoneCard클라스의 매개 하위클라스들은 이 메쏘드를 계승한다. 그러나 전화카드에 따라 이 메쏘드의 구체적실현은 같지 않다. 이것을 실현하기 위하여 하위클라스들은 《전화걸기》메쏘드(performDial())를 다시 정의하고 작성할수 있다. 실례로 자기카드클라스는《전화걸기》메쏘드를 다시 정의할수 있는데《자기카드전화기를 찾아 직접 걸기》를 리용하여 그것을 실현한다. 200카드클라스 역시 PhoneCard로부터 계승한《전화걸기》메쏘드를 다시 정의할수 있다. 즉 《쌍음성주파수전화기를 찾고 우선 카드번호, 비밀번호를 입력한 후 번호를 돌리기》를 리용하여 그것을 실현할수 있다. 모든 클라스에서 전화걸기기능을 실현하는 메쏘드이기만 하면 내용이 같지 않더라도 서로 같은 이름 즉 《performDial》을 공유한다.

이러한 메쏘드의 다중정의는 다형의 중요한 형식으로 된다. 이러한 같은 이름의 서로 다른 메쏘드들은 메쏘드를 호출할 때 어느 클라스의 메쏘드인가를 지정하기만 하면 쉽게 구분할수 있다.(례: MymagCard.performDial()

2) **다형의 둘째 류형**

다형의 둘째 류형은 **재정의**(override)라고 하는데 동일한 클라스에서 같은 이름을 가진 메쏘드를 정의하는 경우이다. 메쏘드의 다중정의와 마찬가지로 이 메쏘드들의 이름이 같은 원인은 그것들의 최종기능과 목적이 같기때문이다. 그러나 서로 다른 구체적인 정황에 부닥칠수 있기때문에 서로 다른 구체적인 내용을 포함하는 메쏘드를 정의하여야 한다. 실례로 클라스가 인쇄기능을 가질것을 요구한다면 인쇄는 범위가 넓은 개념이며 대응하는 구체적인 정황과 조작은 여러가지이다.(례: 실수인쇄, 옹근수인쇄, 문자인쇄, 형가르기인쇄 등) 인쇄기능을 완전히 갖추기 위하여 이 클라스에서 몇개의 print라는 메쏘드를 정의할수 있다. 매개 메쏘드는 다른 메쏘드와 서로 다른 구체적인 인쇄조작을 완성하며 구체적인 인쇄정황을 처리한다. 이때 구체적인 인쇄조작이나 인쇄정황은 체계가 자동적으로 식별하며 그에 상응한 메쏘드를 호출한다.

제3절. 재정의

> **메쏘드의 다중정의에서는 상위클라스와 메쏘드이름, 귀환값과 파라메터목록이 같지만 재정의에서는 한 클라스안에서 메쏘드이름은 같아도 파라메터목록은 다르다.**

메쏘드의 재정의는 다형기술을 실현하는 중요수단이다. 메쏘드의 다중정의와는 달리 재정의는 클라스자체가 이미 가지고있는 같은 이름을 가진 메쏘드를 다시 정의하는것을 말한다.

실례로 200카드를 사용하여 전화를 걸 때 정확한 카드번호, 비밀번호를 이미 입력하여 교환기에 성공적으로 등록하고 련결하였다면 카드번호, 비밀번호를 다시 입력할 필요가 없게 된다. 그러나 교환기에 련결하지 못하였으면 카드번호나 비밀번호를 입력하여야 한다. 그러므로 2개의 같은 이름을 가진 performDial()메쏘드를 정의할수 있다.

실례 4-5

```
Example 4-5 TestOverride.java
1: public class TestOverride
2: {
3:     public static void main(String args[])
4:     {
5:         D200_Card my200 = new D200_Card(12345678, 1234, 50, "200");
```

```
6:          if(my200.performDial(12345678, 1234))
7:              System.out.println("전화건 후 남은 금액:"+my200.getBalance( ));
8:          if(my200.performDial( ))
9:              System.out.println("전화건 후 남은 금액:"+my200.getBalance( ));
10:     }
11:}
12:abstract class PhoneCard
13:{
14:     double balance;
15:
16:     abstract boolean performDial( );
17:     double getBalance( )
18:     {
19:         return balance;
20:     }
21:}
22:abstract class Number_PhoneCard extends PhoneCard
23:{
24:     long cardNumber;
25:     int password;
26:     String connectNumber;
27:     boolean connected;
28:
29:     boolean performConnection(long cn, int pw)
30:     {
31:         if(cn == cardNumber && pw == password)
32:         {
33:             connected = true;
34:             return true;
35:         }
36:         else
37:             return false;
38:     }
39:}
40:class D200_Card extends Number_PhoneCard
41:{
42:     double additoryFee;
43:
44:     D200_Card(long cn, int pw, double b, String c)
45:     {
46:         cardNumber = cn;
47:         password = pw;
```

```
48:         balance = b;
49:         connectNumber = c;
50:     }
51:     boolean performDial( )
52:     {
53:         if(! connected)
54:             return false;
55:         if( balance > (0.5 + additoryFee))
56:         {
57:             balance   -= (0.5 + additoryFee);
58:             return true;
59:         }
60:         else
61:             return false;
62:     }
63:     boolean performDial(long cn, int pass)
64:     {
65:         if(performConnection(cn, pass))
66:             return performDial( );
67:         else
68:             return false;
69:     }
70:     double getBalance( )
71:     {
72:         if(connected)
73:             return balance;
74:         else
75:             return -1;
76:     }
77:}
```

프로그람설명

51-62행에서 형식파라메터를 가지지 않는 D200_Card클라스의 첫번째 performDial( )메쏘드를 정의하였다. 우선 련결을 했는가 하지 않았는가를 검사하고 만일 련결되였다면 balance로부터 통화비용을 공제하고 그렇지 않으면 조작은 실패하여 false를 귀환한다. 63-69행에서 두번째 performDial( )메쏘드를 정의하였다. 이 메쏘드는 긴 옹근수형과 옹근수형의 2가지 형식파라메터를 가지는데 사용자가 입력한 카드번호와 비밀번호를 의미한다. 이 메쏘드는 우선 Number_PhoneCard클라스에서 계승한 performConnection( )메쏘드를 리용한다. 만일 련결이 성공하면 첫번째 performDial( )메쏘드를 리용하여 통화비용을 공제하고 련결표시 connected를 true로 설정하며 그렇지 않으면 조작이 실패하여 false를 귀환한다.

70-76행의 getBalance메쏘드에서도 역시 련결되였는가 되지 않았는가를 검사하며 전화카드의 합법적인 사용자일 때에야 전화카드의 남은 금액을 조사할수 있다. 5행에서는 D200_Card클라스의 객체 my200을 창조하였다. 6행은 이 객체의 두번째 performDial( )메쏘드를 리용하여 전화를 건다. 7행에서는 통화후의 남은 금액을 출력한다. 이때 이미 련결이 성공하였기때문에 8행에서 첫번째 performDial( )메쏘드를 리용하여 두번째 전화를 걸게 된다. 9행에서는 두번째로 전화를 건 후의 남은 금액을 출력한다. 그림 4-5는 실례 4-5의 실행결과이다.

```
D:\Javatextbook\Test>java TestOverride
전화건 후 남은 금액:49.5
전화건 후 남은 금액:49.0

D:\Javatextbook\Test>
```

그림 4-5. 실례 4-5의 실행결과

제4절. 구성자의 계승과 재정의

구성자는 메쏘드의 특수경우이며 상위클라스로부터 계승된다.
또한 재정의가 가능하다.

4.4.1.구성자의 재정의

구성자의 재정의는 동일한 클라스에서 서로 다른 파라메터목록를 가지는 몇개의 구성자가 존재할 때 진행한다.

실례로 D200_Card클라스는 몇개의 구성자를 동시에 정의할수 있으며 이 구성자들은 서로 다른 정황에서 초기화작업을 완성하는데 리용된다.

```
D200_Card( ) // 형식파라메터가 없는 구성자, 임의의 조작을 하지 않는다.
{
}
D200_Card(long cn) // 파라메터의 구성자는 전화카드의 초기화를 진행한다.
{
    cardNumber = cn;
}
D200_Card(long cn, int pw) // 2개의 파라메터를 가진 구성자는 전화카드와
                                비밀번호의 초기화를 진행한다.
{
    cardNumber = cn;
    password = pw;
```

```
}
D200_Card(long cn, int pw, double b) // 3개의 파라메터를 가진 구성자는 카
                드번호와 비밀번호, 금액의 초기화를 진행한다.
{
    cardNumber = cn;
    password = pw;
    balance = b;
}
D200_Card(long cn, int pw, double b, String c) // 4개의 파라메터를 가진 구
                성자는 카드번호와 비밀번호, 금액, 접수번호의 초기화를
                진행한다.
{
cardNumber = cn;
password = pw;
balance = b;
connectNumber = c;
}
```

하나의 클라스에 구성자의 재정의로 인하여 여러개의 구성자가 존재하는 경우 이 클라스객체의 창조시 실제파라메터의 개수, 형과 순서에 따라 어느 구성자를 리용하여 새로운 객체에 대한 초기화작업을 완성하겠는가를 자동적으로 확정한다. 실례로 아래의 3개 명령문은 각각 3개의 서로 다른 구성자를 리용하여 D200_Card클라스의 객체를 창조한다.

```
D200_Card my200 = new D200_Card( ); //파라메터가 없는 구성자의 호출
D200_Card my200 = new D200_Card(12345678,1234); //2개의 파라메터
                                    를 가진 구성자의 호출
D200_Card my200 = new D200_Card(12345678,1234,50); //3개의 파라메터
                                    를 가진 구성자의 호출
```

클라스의 구성자들은 서로 리용될수 있다. 하나의 구성자가 다른 구성자를 리용하는 경우에 예약어 this를 사용할수 있으며 이 호출명령문은 실행가능한 명령문으로서 제일 앞에 놓아야 한다. 실례로 우의 D200_Card클라스의 몇개 구성자는 아래와 같이 고쳐쓸수 있다.

```
D200_Card(long cn) //파라메터가 1개인 구성자는 전화카드번호를 초기화한다.
{
    this( );
    cardNumber = cn;
}
D200_Card(long cn, int pw) // 파라메터가 2개인 구성자는 카드번호와 비밀
                    번호를 초기화한다.
{
    this(cn);
```

```
        password = pw;
    }
    D200_Card(long cn, int pw, double b) // 파라메터가 3개인 구성자는 카드번
                                          호와 비밀번호, 금액을 초기화한다.
    {
        this(cn, pw);
        balance = b;
    }
    D200_Card(long cn, int pw, double b, String c) // 파라메터가 4개인 구성자
                                          는 카드번호와 비밀번호, 금액, 접수번호를 초기화한다.
    {
        this(cn, pw, b);
        connectNumber = c;
    }
```

4.4.2.구성자의 계승

하위클라스는 상위클라스의 구성자를 계승할수 있으며 구성자의 계승은 아래의 원칙에 따라야 한다.

· 하위클라스는 상위클라스의 파라메터를 포함하지 않은 구성자를 무조건 계승한다.

· 하위클라스자체가 구성자를 가지지 않으면 그것은 상위클라스에서 계승한 파라메터를 포함하지 않은 구성자를 자기의 구성자로 한다. 만일 하위클라스자체가 구성자를 정의하였으면 새로운 객체창조시 먼저 자기 상위클라스를 계승한 파라메터를 포함하지 않은 구성자를 실행하며 다음에 자기의 구성자를 다시 실행한다.

· 파라메터를 가지고있는 상위클라스의 구성자에 대하여 하위클라스는 자기의 구성자에서 super예약어를 사용하여 호출할수 있으며 이 호출명령문은 반드시 하위클라스구성자에서 제일 앞에 놓아야 한다.

아래에서 몇개의 례제들을 보자. 상위클라스 Number_PhoneCard가 5개의 구성자를 가진다고 가정한다.

```
    Number_PhoneCard( )
    {
    }
    Number_PhoneCard(long cn)
    {
        cardNumber = cn;
    }
    Number_PhoneCard(long cn, int pw)
    {
```

```
        cardNumber = cn;
        password = pw;
    }
    Number_PhoneCard(long cn, nt pw, double b)
        cardNumber = cn;
        password = pw;
        balance = b;
    }
    Number_PhoneCard(long cn, int pw, double b, String c)
    {
        cardNumber = cn;
        password = pw;
        balance = b;
        connectNumber = c;
    }
```

하위클라스 D200_Card의 구성자는 다음과 같은 몇가지 방법으로 설계할수 있다.

· 자기의 구성자를 전문적으로 정의하지 않는다. 이런 상태에서 200전화카드객체를 창조할 때 체계가 자동적으로 호출하는것은 상위클라스 Number_PhoneCard의 파라메터를 포함하지 않은 구성자이다.

· 자기의 구성자를 정의하는데 상위클라스의 파라메터를 포함하지 않은 구성자를 먼저 호출한다. 이 상태에서 하위클라스는 상위클라스구성자가 정의한 초기화조작에 기초하여 하위클라스자체의 초기화조작을 진행한다.

```
    D200_card(long cn, int pass, double b, double a)
    {
        super(cn, pass, b); // 상위클라스의 구성자를 호출하여 매 구역에 대하
                               여 초기값을 설정한다.
        additoryFee = a;   // 새로운 파라메터를 리용하여 부가비를 초기화한다.
    }
```

· 하위클라스에서 구성자의 재정의를 실현한다. 이 경우는 여러 계층의 객체초기화요구를 만족시킬수 있다.

```
    D200_Card(long cn, int pw, double a)
    {
        super(cn, pw); // 상위클라스의 구성자를 호출하여 초기값을 설정한다.
        additoryFee = a; // 새로운 파라메터를 리용하여 부가비를 초기화한다.
    }
    D200_Card(long cn, int pw, double d, String c, double a)
    {
```

```
        super(cn, pw, d, c); // 상위클라스의 구성자를 호출하여 초기값을 설정
                              한다.
        additoryFee = a; // 새로운 파라메터를 리용하여 부가비를 초기화한다.
    }
```

실례 4-6은 구성자의 계승과 재정의규칙을 함께 사용한것이다.

실례 4-6

```
Example 4-6 ConstructorOverride.java
1: public class ConstructorOverride
2: {
3:      public static void main(String args[])
4:      {
5:          D200_Card my200 = new D200_Card(12345678, 1234, 50.0, "200", 0.1);
6:          System.out.println(my200.toString( ));
7:      }
8: }
9: abstract class PhoneCard
10:{
11:     double balance;
12:
13:     abstract boolean performDial( );
14:     double getBalance( )
15:     {
16:         return balance;
17:     }
18:}
19:abstract class Number_PhoneCard extends PhoneCard
20:{
21:     long cardNumber;
22:     int password;
23:     String connectNumber;
24:     boolean connected;
25:
26:     Number_PhoneCard( )
27:     {
28:     }
29:     Number_PhoneCard(long cn)
30:     {
31:         this( );
```

```
32:         cardNumber = cn;
33:     }
34:     Number_PhoneCard(long cn, int pw)
35:     {
36:         this(cn);
37:         password = pw;
38:     }
39:     Number_PhoneCard(long cn, int pw , double b)
40:     {
41:         this(cn, pw);
42:         balance = b;
43:     }
44:     Number_PhoneCard(long cn, int pw, double b, String c)
45:     {
46:         this(cn, pw, b);
47:         connectNumber = c;
48:     }
49:     boolean performConnection(long cn, int pw)
50:     {
51:         if(cn == cardNumber && pw == password)
52:         {
53:             connected = true;
54:             return true;
55:         }
56:         else
57:             return false;
58:     }
59:}
60:class D200_Card extends Number_PhoneCard
61:{
62:     double additoryFee;
63:
64:     D200_Card(long cn, int pw, double a)
65:     {
66:         super(cn, pw);
67:         additoryFee = a;
68:     }
69:     D200_Card(long cn, int pw, double b, double a)
70:     {
71:         super(cn, pw, b);
72:         additoryFee = a;
73:     }
```

```
74:    D200_Card(long cn, int pw, double b, String c, double a)
75:    {
76:        super(cn, pw, b, c);
77:        additoryFee = a;
78:    }
79:    boolean performDial( )
80:    {
81:        if(! connected)
82:            return false;
83:        if(balance > (0.5 + additoryFee))
84:        {
85:            balance -= (0.5 + additoryFee);
86:            return true;
87:        }
88:        else
89:            return false;
90:    }
91:    boolean performDial(long cn, int pass)
92:    {
93:        if(performConnection(cn, pass))
94:            return performDial( );
95:        else
96:            return false;
97:    }
98:    double getBalance( )
99:    {
100:       if(connected)
101:           return balance;
102:       else
103            return -1;
104:   }
105:   public String toString( )
106:   {
107:       return("전화카드접수번호:" + connectNumber
108:               + "\n전화카드번호:" + cardNumber
109:               + "전화카드비밀번호:" + password
110:               + "\n카드의  금액:" + balance
111:               + "\n통화부가비:" + additoryFee);
112:   }
113:}
```

프로그람설명

26-48행에서 재정의를 사용하여 Number_PhoneCard클라스의 5개 구성자를 정의하였다. 64-78행에서 계승과 재정의를 사용하여 D200_Card클라스의 3개의 구성자를 정의하였다. 5행에서는 D200_Card클라스의 객체 my200을 창조하여 모든 마당에 대하여 초기화를 진행하였다.

그림 4-6은 실례 4-6의 실행결과이다.

```
D:\Javatextbook\Test>java ConstructorOverride
전화카드접수번호:200
전화카드번호:12345678전화카드비밀번호:1234
카드의 금액:50.0
통화부가비:0.1

D:\Javatextbook\Test>
```

그림 4-6. 실례 4-7의 실행결과

제5절. 패키지

- 패키지는 서로 접근할수 있고 일정한 련관을 가지는 클라스들의 모임이다.
- 패키지창조는 현재등록부안에서 하나의 하위등록부를 창조하는것이다.
- 패키지의 인용은 예약어 import를 리용한다.

객체지향기술을 리용하여 실제적인 체계를 개발할 때 보통 많은 클라스들을 정의하여 공동작업을 진행하여야 한다. 이 클라스들을 보다 효과적으로 관리하기 위하여 Java에서는 패키지의 개념을 도입하였다. 등록부가 매 파일들을 함께 관리하는것과 같이 Java의 패키지는 매 클라스들을 함께 조직하며 프로그람기능을 명백히 하고 구조가 뚜렷하게 한다. 특히 패키지를 사용하면 서로 다른 프로그람들사이에서 클라스들의 반복사용을 실현하는데서 아주 유리하다.

패키지는 일반적으로 클라스들의 모임이다. 같은 패키지에 있는 클라스들이 포함이나 계승관계를 가질것을 요구하지는 않는다. 같은 패키지의 클라스는 기정인 상황에서 서로 접근할수 있으므로 프로그람작성과 관리를 편리하게 하기 위해 보통 함께 작업할것을 요구하는 클라스들을 같은 패키지에 놓는다. 실례로 클라스 PhoneCard, Number_phoneCard와 D200_Card 등을 같은 패키지에 놓을수 있다.

4.5.1. 패키지창조

기정으로 체계는 매개 java원천파일에 대하여 이름이 없는 패키지를 작성할수 있으며 이 java파일에서 정의한 모든 클라스는 이 패키지에 속하고 그것들사이에는 서로 공개마당이나 메쏘드를 인용할수 있다. 그러나 이름이 없는 패키지안의 클라스들은 다른 패키지의 클라스에 의하여 인용되고 반복사용될수 없다. 이러한 문제를 해결하기 위하여 이름을 가지는 패키지를 창조하여야 한다. 패키지창조는 예약어 package를 사용하여야 하며 java파일의 제일 앞에 있어야 한다.

```
package 패키지명;
```

이 명령문을 리용하면 지정한 이름을 가진 패키지를 창조할수 있으며 현재 java파일의 모든 클라스는 이 패키지에 놓이게 된다. 아래의 실례를 고찰해보자.

```
package CardClasses;
package CardSystem.CardClasses;
```

사실상 패키지창조는 현재등록부안에서 하나의 보조등록부를 창조하며 이 패키지에 포함되는 모든 클라스들을 파일로 보관해놓는다. 우의 두번째 패키지창조명령문의 기호 《.》은 등록부구분부를 의미하며 이 명령문에 의해 두개의 등록부가 창조된다. CardSystem은 현재등록부의 보조등록부이고 CardClasses는 CardSystem의 보조등록부이며 현재패키지의 모든 클라스들은 이 등록부안에 놓인다.

아래의 례제는 앞에서 사용한 PhoneCard클라스와 그의 하위클라스 Number_PhoneCard, D200_Card를 CardClasses라는 패키지로 묶는다.

```
package CardClasses; // 패키지의 창조
abstract class PhoneCard
{
    double balance;
    abstract boolean performDial( );
    double getBalance( )
    {
        return balance;
    }
}
abstract class Number_PhoneCard extends PhoneCard
{
    long cardNumber;
    int password;
    String connectNumber;
    boolean connected;
    boolean performConnection(long cn, int pw)
    {
        if(cn == cardNumber && pw == password)
```

```
            {
                connected = true;
                return true;
            }
            else
                return false;
        }
    }
    class D200_Card extends Number_PhoneCard
    {
        double additoryFee;
        double balance;
        boolean performDial( )
        {
            if(balance > (0.5 + additoryFee))
            {
                balance -= (0.5 + additoryFee);
                return true;
            }
            else
                return false;
        }
    }
```

프로그람설명

이 프로그람은 현재등록부에서 보조등록부 CardClasses를 창조하여 프로그람에서 정의한 3개의 클라스 PhoneCard, Number_PhoneCard와 D200_Card가 생성한 3개의 대응하는 바이트코드파일 PhoneCard.class, Number_PhoneCard.class, D200_Card.class들을 이 등록부에 보관한다. 만일 이 프로그람에서 다른 클라스를 더 정의하였다면 그것들 역시 같은 등록부안에 놓이게 된다.

4.5.2. 패키지의 인용

1) 패키지이름과 클라스이름을 앞붙이로 사용

하나의 클라스가 다른 클라스를 인용하려면 이 클라스를 계승하거나 이 클라스의 객체를 창조한 다음 그의 마당을 사용하고 메쏘드를 호출하여야 한다. 같은 패키지의 다른 클라스에 대하여서는 사용해야 하는 속성이나 메쏘드이름앞에 앞붙이로서 클라스이름을 덧붙여야 한다. 다른 패키지의 클라스에 대하여서는 클라스이름앞에 패키지이름을 앞붙이로 덧붙여야 한다.

```
CardClasses.D200_Card my200 = new CardClasses.D200_Card(12345678,1234);
System.out.println(my200.toString( ));
```

2) 사용해야 할 클라스의 적재

만일 우의 방법대로 한다면 D200_Card클라스가 출현할 때마다 반드시 패키지이름을 앞붙이로 추가해야 하므로 프로그람작성에서 사용하기가 아주 불편하다. 이것을 해결하기 위하여 프로그람파일의 시작부분에서 import문을 리용하여 사용해야 하는 전체 클라스를 현재프로그람에 인입한다.

```
import CardClasses.D200_Card; // 프로그람시작에서 다른 패키지의 클라스를 인입
```

이 명령문다음부터는 직접 다음과 같이 쓸수 있다.

```
D200_Card my200 = new D200_Card(12345678, 1234);
```

우의 방법은 import문을 리용하여 다른 패키지의 클라스를 적재하고있다. 어떤 경우에는 import문을 리용하여 패키지의 모든 클라스를 프로그람에 적재할수 있다.

```
import CardClasses.*;
import java.awt.*;
```

3) CLASSPATH

패키지는 코드를 조직하는 유효한 수단이며 패키지이름은 실제상 프로그람에서 사용하는 클라스파일(확장자가 class인 파일)이 들어있는 등록부의 위치를 가리킨다. 이 클라스파일이 들어있는 등록부의 위치를 환경변수 CLASSPATH를 리용하여 지정할수 있다. CLASSPATH는 DOS조작체계에서의 PATH와 류사하며 그것은 모든 기정인 클라스바이트코드파일의 경로를 가리킨다. 어떤 프로그람이 자기가 사용해야 할 클라스파일을 찾을수 없을 때 체계는 자동적으로 CLASSPATH환경변수가 지적하는 경로에 가서 찾는다.

CLASSPATH환경변수는 체계의 AUTOEXEC.BAT파일편집이나 련관있는 DOS방법을 통하여 설정할수 있다. 실례로 아래의 명령문

```
SET CLASSPATH=.;c:\jdk1.2\lib;c:\jdk1.2\lib.classes.zip
```

는 CLASSPATH를 현재의 등록부 c:\jdk1.2\lib와 c:\jdk1.2\lib\lib\classes.zip로 설정한다.

Java Application프로그람에 대하여서도 파라메터를 설정하여 클라스파일경로를 지정할수 있다. 실례로 JDK의 Java해석기 java.exe에는 개페기파라메터 -classpath가 있으며 Visual J++의 Java해석기 jview.exe에는 파라메터 -cp가 있다. 해석실행해야 할 test.class파일이 현재등록부에 있는것이 아니라 C구동기의 TEMP등록부에 있다고 가정하면 아래의 지령행문을 사용할수 있다.

```
java test -classpath c:\temp
```

제6절. 대면

- 대면은 추상클라스의 특수한 형태이다.
- 대면은 메쏘드와 상수만으로 구성된다.
- 다중계승을 실현할수 있다.
- 대면을 실현하는 클라스는 대면으로부터 계승한 추상메쏘드들을 모두 구현하여야 한다.

Java의 대면은 패키지와 류사하며 응용프로그람의 매 클라스를 조직하고 그것들의 호상관계구조를 조절하는데 리용된다. 다시말하여 대면은 클라스들사이의 다중계승기능을 실현하는데 쓰이는 구조이다.

4.6.1. 대면의 개요

Java에서 대면은 문법상 클라스와 류사한데 그것은 몇개의 추상메쏘드와 벡토르를 정의하여 하나의 속성모임을 형성한다. 이 속성모임은 보통 어떤 기능에 대응되며 그의 작용은 클라스와 류사한 다중계승의 기능을 실현할수 있다.

다중계승이라는것은 하나의 하위클라스가 하나이상의 직접적인 상위클라스를 가질수 있다는것을 의미하며 이 하위클라스는 모든 직접적인 상위클라스의 성원을 계승할수 있다. 어떤 객체지향언어 례하면 C++는 다중계승의 문법지원을 제공하지만 Java에서는 프로그람구조를 간단화할것을 고려하여 나왔으므로 클라스들사이의 다중계승을 지원하지 않고 단일계승만을 지원한다. 즉 하나의 클라스는 한개의 직접적인 상위클라스만을 가진다. 그러나 실제문제를 해결하는 과정에 단일계승은 일부 복잡한 문제들을 완전히 표현할수 없으며 다중적인 기구들이 보조적으로 요구되게 된다.

Java프로그람에서는 클라스계승구조를 보다 더 합리적으로 조직하고 실제문제의 본질에 더 부합되게 하기 위하여 특정한 기능을 완성하는데 쓰는 일부 속성을 상대적으로 독립인 속성모임으로 조직할수 있게 하였다. 이런 특정한 기능의 실현을 요구하는 클라스이면 이 속성모임을 계승하고 클라스내에서 그것을 사용할수 있다. 이 속성모임이 바로 대면이다. 도형사용자대면부프로그람에서 사용한 ActionListener가 바로 체계가 정의한 대면으로서 그것은 동작사건을 감시하고 처리하는 기능을 나타내며 여기에 추상메쏘드를 포함하고있다. 즉

```
public void actionPerformed(ActionEvent e);
```

동작사건(례: 단추의 찰칵, 본문칸에서의 되돌이 등)을 처리하려는 모든 클라스들은 반드시 ActionListener대면이 정의한 기능을 가져야 한다. 구체적으로 말하여 이 대면을 실현하고 actionPerformed()메쏘드를 다중정의하여야 한다.

중요한것은 Java에서 클라스는 이런 대면이 정의한 기능을 얻는다는 의미이지 이 대면의 속성과 메쏘드를 직접 계승한다는것은 아니다. 그러므로 대면의 속성들은

모두 상수이며 대면의 메쏘드들은 모두 메쏘드본체를 가지지 않는 추상메쏘드이다. 따라서 이 클라스들에서 대면의 매 추상메쏘드의 메쏘드본체를 구체적으로 정의해야 한다. Java에서 보통 대면기능에 대한 《계승》을 《실현》이라고 말한다.

4.6.2. 대면의 선언

Java에서 대면을 선언하는 문법적구조는 다음과 같다.

```
[public] interface 대면이름 [extends 상위대면목록]
{   //대면본체
    [public] [static] [final] 마당형 마당이름 = 상수값; // 상수마당선언
    [public] [abstract] [native] 귀환값 메쏘드이름(파라메터목록) [throw 례외목록];
                                                  //추상메쏘드선언
}
```

우의 문법정의에서 볼수 있는것처럼 대면정의는 클라스정의와 아주 류사하다. 대면은 상수와 추상메쏘드로 구성된 특수클라스이다. 클라스는 하나의 상위클라스밖에 가질수 없지만 대면은 몇개의 대면을 동시에 실현할수 있다. 이러한 상황에서 만일 대면을 특수한 클라스로 리해하면 이 클라스가 대면을 리용한다는것은 실제상 많은 상위클라스를 계승한것처럼 볼수 있다. 즉 다중계승을 실현하는것으로 된다. class가 클라스를 선언하는 예약어인것처럼 interface는 대면선언의 예약어이다. 클라스정의와 같이 대면을 선언할 때 접근조종부를 정의하여야 하는데 접근조종부로는 오직 public만을 리용할수 있다. public를 리용하여 장식한 대면은 공개대면이며 모든 클라스와 대면에 의하여 사용될수 있다. 그러나 public를 가지지 않는 대면은 같은 패키지의 다른 클라스와 대면에 의하여서만 리용될수 있다.

클라스와 마찬가지로 대면 역시 계승성을 가진다. 대면을 정의할 때 extends예약어를 통하여 새로운 대면이 이미 존재하는 어떤 상위대면의 파생대면이라는것을 선언할수 있으며 그것은 상위대면의 모든 속성과 메쏘드를 계승하게 된다. 클라스의 계승과 다른것은 대면은 하나이상의 상위대면을 가질수 있다는것이며 그것들사이는 반점으로 구별하여 상위대면목록을 형성한다. 이때 새 대면은 모든 상위대면의 속성과 메쏘드를 계승한다.

대면본체의 선언은 대면을 정의하는 기본부분이다. 대면본체는 두 부분으로 구성되는데 한 부분은 속성에 대한 선언이고 다른 한 부분은 메쏘드에 대한 선언이다. 앞에서 이미 지적한것처럼 대면의 메쏘드는 모두 abstract를 리용하여 장식한 추상메쏘드이다. 대면에서는 이 추상메쏘드들의 이름과 귀환값, 파라메터목록만 줄수 있으며 메쏘드본체를 정의할수 없다.

대면에서의 모든 속성들은 반드시 public static final이여야 하는데 이 장식부를 꼭 써주지 않아도 된다. 마찬가지로 대면의 모든 메쏘드는 반드시 public abstract여야 하며 이 장식부를 써주지 않아도 된다. 대면에서 메쏘드의 본체는 Java언어로 작

성할수도 있고 다른 언어로 작성할수도 있다. 메쏘드의 본체를 다른 언어로 작성하는 경우 대면메쏘드는 native장식부를 리용하여야 한다.

Java의 체계클라스서고에는 대면을 사용하는 적지 않은 실례들이 있다. 아래에 체계가 정의한 대면 DataInput의 정의를 보여준다.

```
public interface java.io.DataInput
{
    public abstract boolean readBoolean( );          //론리형자료읽기
    public abstract byte readByte( );                 //바이트형자료읽기
    public abstract char readChar( );                 //문자형자료읽기
    public abstract double readDouble( );             //배정확도자료읽기
    public abstract float readFloat( );               //류동소수점수자료읽기
    public abstract void readFully(byte b[]);         //자료전부를 읽고 바이트
                                                        배렬 b에 보관
    public abstract void readFully(byte b[],int off,int len); //자료전부를 읽고
                                                        일정한 위치에 보관
    public abstract int readInt( );                   //옹근수형자료읽기
    public abstract String readLine( );               //한행읽기
    public abstract long readLong( );                 //긴옹근수형자료읽기
    public abstract short readShort( );               //짧은옹근수형자료읽기
    public abstract int readUnsignedByte( );          //부호없는 바이트형자료읽기
    public abstract int readUnsignedShort( ); //부호없는 짧은 옹근수형자료읽기
    public abstract String readUTF( );                //UTF자료읽기
    public abstract int skipBytes(int n);     //위치를 읽고 n개 바이트를 뛰여넘기
}
```

이 대면에서는 자료형에 따라 자료를 읽어들이는 메쏘드들을 정의하였다. Java에서 가장 기본적인 입출력은 흐름식입출력이다. 즉 자료의 본래의 의미와 실제적인 자료형을 리해하지 않고 단지 자료를 1렬순서의 비트흐름으로 보고 읽어들이거나 출력할뿐이다. 물론 이것은 아주 저급한 입출력방식이다. 프로그람작성을 간단히 하기 위하여 Java에서는 우에서 서술한 DataInput대면과 DataOutput대면을 정의하고있다. 이 두개의 대면에서 자료형에 따라 자료를 읽어들이는 여러가지 메쏘드들을 정의하고 있으며 이것들은 각각 《자료형에 따르는 입력》과 《자료형에 따르는 출력》의 기능을 나타낸다. 만일 입출력과 련관있는 클라스를 써서 이 두개의 대면을 실현하면 이 클라스들은 비교적 고급한 자료형에 따르는 읽기, 쓰기기능을 가지게 된다.

4.6.3. 대면의 실현

대면의 선언에서 추상메쏘드만을 주었으므로 대면이 지정하는 기능을 실현하여야 한다. 어떤 대면의 추상메쏘드에 대하여 명령문들을 작성하고 실제한 메쏘드본체를 정의하는것을 대면의 실현이라고 한다.

어떤 클라스가 대면을 실현할 때 다음의 문제들에 주의하여야 한다.

· 클라스의 선언부분에서 implements예약어를 리용하여 이 클라스가 어느 대면을 실현하는가를 선언한다.

· 만일 어떤 대면을 실현하는 클라스가 abstract클라스가 아니라면 클라스의 정의부분에서 반드시 대면을 지정하는 모든 추상메쏘드를 실현하여야 한다. 즉 모든 추상메쏘드에 대하여 메쏘드본체를 정의하여야 한다. 또한 메쏘드머리부분은 대면의 정의와 완전히 일치해야 한다. 즉 완전히 같은 귀환값과 파라메터목록을 가져야 한다.

· 만일 어떤 대면을 실현하는 클라스가 abstract클라스이면 이 대면의 모든 메쏘드들을 실현할수 없다. 그러나 이 추상클라스의 추상이 아닌 하위클라스에 대하여서는 상위클라스가 실현하는 대면의 모든 추상메쏘드들에 대한 실제의 메쏘드본체를 정의해야 한다. 이 메쏘드본체들은 추상인 상위클라스에서부터 나올수도 있고 하위클라스 자체에서부터 나올수도 있다. 그러나 실현하지 못한 대면메쏘드가 존재하는것은 허용하지 않는다. 그것은 추상이 아닌 클라스에서 추상메쏘드가 존재할수 없기때문이다.

· 클라스가 어떤 대면의 추상메쏘드를 실현하려면 완전히 같은 메쏘드머리부를 사용하여야 한다. 만일 실현한 메쏘드가 추상메쏘드와 이름은 같으나 서로 다른 파라메터목록을 가진다면 새로운 메쏘드를 재정의할뿐이지 이미 있는 추상메쏘드를 실현하는것은 아니다.

· 대면의 추상메쏘드의 접근조종부는 이미 public로 지정되여있으므로 클라스가 메쏘드를 실현할 때 반드시 public장식부를 사용하여야 한다. 그렇지 않으면 체계는 대면에서 정의하는 메쏘드의 접근조종범위가 축소되였다고 경고한다.

실례 4-7

```
Example 4-7 ImplementActionListener.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class ImplementActionListener extends Applet implements ActionListener
6: {
7:      TextField password =   new   TextField("비밀번호");
8:      Button btn = new Button("감추기");
9:      public void init( )
```

```
10:    {
11:        add(password);
12:        add(btn);
13:        btn.addActionListener(this);
14:    }
15:    public void actionPerformed(ActionEvent e)
16:    {
17:        password.setEchoChar('*');
18:        password.selectAll( );
19:    }
20:}
```

프로그람설명

실례 4-7의 ImplementActionListener클라스는 ActionListener대면을 실현하고있다. ActionListener대면의 정의는 아래와 같다.

```
public abstract interface ActionListener extends EventListener
{
    public void actionPerformed(ActionEvent e);
}
```

우에서 EventListener는 ActionListener의 상위대면이며 actionPerformed( )메쏘드는 기정으로 추상메쏘드이다.

실례 4-7에서 5행은 주클라스 ImplementActionListener가 ActionListener대면을 실현한다는것을 선언한다. 15-19행에서는 ActionListener대면에서 정의한 actionPerformed( )추상메쏘드를 다시 정의하고있다. 그의 작용은 사용자가 《감추기》단추를 찰칵하는 동작에 응답하는것이며 본문마당 password의 문자를 '*'로 바꾸고 여기의 문자를 선택하는것이다. 그림 4-7은 실례 4-7의 실행결과이다.

그림 4-7. 실례 4-7의 실행결과

제5장. 도구클라스

이 장에서는 먼저 Java프로그람작성에서 일상적으로 사용하는 도구클라스들을 소개한다. 여기에서 Java의 언어기초클라스서고, Applet프로그람을 서술한다.

제1절. 기초클라스서고

기초클라스서고: Object클라스, 기본자료형클라스, Math클라스, System클라스

5.1.1. Object클라스

Object클라스는 Java프로그람의 모든 클라스의 직접 또는 간접적인 상위클라스이며 클라스서고에서 모든 클라스의 상위클라스이다. 모든 클라스들은 Object클라스로부터 파생한것이며 Object클라스는 모든 Java클라스의 공개속성을 포함하고있다. 여기서 비교적 중요한 메쏘드들은 아래와 같다.

protected object clone( ) // 현재의 복사객체를 생성하고 이 복사객체를 귀환한다.
public boolean equals(Object obj) // 2개의 객체가 같은가 같지 않은가를 비교하고 같으면 true를 귀환한다.
public final Class getClass( ) // 현재객체가 속하는 클라스정보를 얻고 Class객체를 귀환시킨다.
protected void finalize( ) // 현재객체를 회수할 때 완성해야 할 정리작업을 정의한다.
public String toString( ) // 현재객체자체의 련관정보를 문자렬로 귀환한다.

Object클라스는 모든 Java클라스의 상위클라스이고 또한 임의의 류형과 통합할수 있으므로 어떤 경우에는 그것이 형식파라메터로 되기도 한다. 실례로 우의 equals( ) 메쏘드이다. 이 형식파라메터는 Object류형의 객체이며 이렇게 하여 임의의 Java클라스가 자기 객체와 직접 서로 비교하는 조작을 정의할수 있다. Object클라스의 사용은 이 메쏘드의 실제파라메터가 임의의 류형의 객체를 사용할수 있게 하며 그렇게 하여 메쏘드의 적용범위를 확대하였다.

5.1.2. 기본자료형클라스

앞에서 이미 Java의 기본자료형인 int, double, char, long 등을 소개하였다. 이 기본자료형을 리용하여 간단한 변수와 속성을 정의하는것은 편리하지만 일련의 자료형변수의 변환과 조작 즉 하나의 문자렬을 옹근수나 류동소수점수로 변환해야 한다면 자료형클라스의 상응한 메쏘드를 사용하여야 한다.

자료형클라스는 기본자료형과 밀접한 련관이 있으며 매개 자료형클라스에는 하나의 기본자료형이 대응하고 이름도 역시 이 기본자료형의 이름과 류사하다.(표 5-1)

다른점은 자료형클라스는 클라스이고 자기의 메쏘드를 가진다는것이다. 이 메쏘드들은 기본적으로 그것에 대응하는 기본자료형변수을 조작하고 처리하는데 리용된다.

표 5-1. 자료형클라스와 그에 대응하는 기본자료형

자료형클라스	기본자료형
Boolean	boolean
Character	char
Double	double
Float	float
Integer	int
Long	long

integer클라스를 실례로 하여 이 메쏘드와 그의 작용을 보기로 하자. 다른 자료형클라스에서의 메쏘드와 그의 작용도 integer클라스와 비슷하다. integer클라스에서는 MAX_VALUE, MIN_VALUE속성과 한개의 메쏘드를 정의하고있다. MAX_VALUE와 MIN_VALUE의 2개마당은 각각 int클라스형변수의 최대값과 최소값을 규정한다.

구성자 public Integer(int value)와 public Integer(String s)는 각각 기본자료형 int변수와 문자렬객체를 리용하여 한개의 Integer객체를 생성할수 있다.

자료형의 변환메쏘드는 각각 현재객체에 대응하는 int변수를 다른 기본자료형의 변수로 변환하고 변환후의 값을 귀환한다.

```
public double doubleValue( )
public int intValue( )
public long longValue( )
```

문자렬과 int변수를 서로 변환하는 메쏘드:

public String toString( )메쏘드는 현재의 integer객체에 대응하는 int변수를 문자렬로 변환한다.

public static int parseInt(String s)메쏘드는 클라스의 메쏘드로서 integer객체를 창조할 필요가 없이 문자렬을 int변수로 변환한다. 아래의 명령문은 문자렬 "123"을 옹근수 123으로 변환하여 변수 i에 값주기한다.

```
int i=Integer.parseInt("123");
```

public static Integer valueOf(String s)메쏘드 역시 클라스의 메쏘드로서 그것은 문자렬을 Integer객체로 변환하며 이 객체에 대응하는 int수값은 문자렬이 표시하는 수값과 일치하다. 아래의 명령문은 우선 valueOf( )메쏘드를 사용하여 문자렬을 Integer객체로 변환하고 이 객체의 intValue( )메쏘드를 호출하여 그에 대응하는 int수

값을 귀환시킨다. 이 작용은 실지로 우의 명령문과 완전히 같다. 그러나 처리과정에 림시 Integer의 객체를 생성한다는것이 좀 차이난다.

```
int i=Integer.valueOf("123").intValue( );
```

어떤 자료형 례를 들어 double과 float는 parseInt( )메쏘드를 가지지 않으므로 valueOf( )메쏘드를 리용하여 문자렬을 수값자료로 변환할수 있다. 실례로

```
float f=Float.valueOf("12.3").floatValue( );
```

5.1.3. System클라스

System클라스는 기능이 강하고 아주 유용한 특수한 클라스로서 표준입출력, 실행시의 체계정보 등 중요도구들을 제공하고있다. 이 클라스는 실례화할수 없다. 즉 System클라스의 객체를 창조할수 없으므로 그것의 속성과 메쏘드는 static이며 인용시 System을 앞붙이로 하여야 한다.

System클라스를 리용하여 표준입출력을 해보자.

System클라스의 속성에는 3가지가 있다. 즉 체계의 표준입력, 표준출력, 표준오유출력이다.

```
public static PrintStream err;
public static InputStream in;
public static PrintStream out;
```

이 3개의 속성을 사용하여 Java프로그람은 표준입력으로부터 자료를 읽어들이고 표준출력에 자료를 쓸수 있다.

```
char c=System.in.read( ); //표준입력으로부터 1byte의 정보를 읽어들여 문
                                    자변수에 귀환시킨다.
System.out.println("Hello!Guys,"); //표준출력에 문자렬을 출력한다.
```

일반적으로 표준입력은 건반이며 표준출력이나 표준오유출력은 화면이다.

System클라스의 메쏘드를 리용하여 체계정보를 얻거나 체계조작을 완성할수 있다. System클라스는 Java를 실행하는 체계와 호상조작을 진행하는데 쓰이는 메쏘드를 제공하고있다. 그것들을 리용하면 Java해석기나 하드웨어기반의 체계파라메터정보를 얻을수 있고 또한 실행체계에 직접 지령하여 조작체계급의 체계조작을 완성할수도 있다. 아래에 몇가지 System클라스메쏘드를 보여준다.

public static long currentTimeMillis( ): 1970년 1월1일 0시부터 현재 체계시간까지의 미리초를 얻으며 보통 두 사건발생의 시간차를 비교하는데 쓴다.

public static void exit(int status): 프로그람의 사용자토막처리가 완전실행되기 전에 Java가상기계는 강제적으로 실행상태를 탈퇴시키며 실행정보 status를 가상기계를 실행시키는 조작체계에 귀환시킨다.(실례로 system.exit(0))

public static void gc( ): Java가상기계의 페품회수기능을 강제리용하여 기억기에서 이미 잃어버린 휴지객체가 차지하고있는 공간을 수집하고 다시 리용할수 있게 해준다.

제2절. Applet클라스

- Applet프로그람작성은 반드시 java.applet패키지의 체계클라스 Applet를 리용하여야 한다.
- Applet가 열람기로부터 자동호출하는 주요메쏘드인 init( ), start( ), stop( ), destroy( )는 각각 Applet의 초기화, 기동, 림시정지, 소멸까지의 생명주기의 단계에 대응한다.

Applet프로그람은 아주 중요한 Java프로그람으로서 인터네트에서 작업하는 열람기상의 프로그람이다. Applet프로그람작성은 반드시 java.applet패키지에서의 체계클라스 Applet를 리용하여야 한다. 이 절에서는 Applet클라스와 Applet프로그람의 련관내용들을 소개한다.

5.2.1. Applet의 기본작업원리

Applet는 일종의 특수한 Java프로그람이다. 해석형언어로서 Java의 바이트코드프로그람은 전문적인 해석기를 통하여 집행된다. Java Application에 대하여 말한다면 이 해석기는 독립적인 체계프로그람이다.(례: JDK의 java.exe, VJ++의 jview.exe 등) 한편 Java Applet에 대하여 말한다면 이 해석기는 인터네트의 열람기프로그람이며 보다 정확히는 Java호환의 인터네트열람기이다.

Applet의 기본작업원리는 다음과 같다. 번역한 바이트코드파일(.class)을 특정한 WWW봉사기상에 보존하고 이 바이트코드파일이름을 삽입한 HTML파일을 동일한 또는 다른 WWW봉사기에 보존한다. 어떤 열람기가 봉사기에 Applet를 삽입한 HTML파일을 내리적재할것을 요구할 때 이 파일은 WWW봉사기로부터 의뢰기측에 내리적재되며 WWW열람기에서 HTML의 각종 꼬리표를 해석한다. 그의 약속에 따라 파일에서의 정보를 일정한 격식으로 사용자화면에 현시한다. 열람기는 HTML파일의 <Applet>꼬리표에 있는 바이트코드를 WWW봉사기로부터 내리적재한 다음 열람기자체가 가지고있는 Java해석기를 리용하여 이 바이트코드를 실행한다.

어떤 의미에서 Applet는 부품이나 조종부품과 류사하다. 독립적인 응용프로그람과 달리 Applet프로그람이 실현하는 기능은 완전하지 않으므로 열람기에서 이미 실현한 기능과 함께 결합해야 완전한 프로그람을 구성할수 있다. 실례로 Applet는 자기의 기본흐름틀을 확립하지 못하므로 열람기는 그것에 대해 자동적으로 기본흐름틀을 세우고 유지한다. 한편 자기의 전문적인 도형사용자대면부를 꼭 가질것을 요구하지 않으며 열람기가 이미 가지는 도형사용자대면부를 직접 리용할수 있다. Applet가 작성해야 할것은 열람기가 발송하는 통보문이나 사건을 접수하는것이다.(례: 마우스이동, 건누르기 등) 그밖에 열람기와의 합작과정을 협조하기 위하여 Applet는 열람기에 의한 메쏘드들을 가지고있다.

5.2.2. Applet클라스

Applet프로그람은 구성상 반드시 한개의 사용자클라스를 창조하여야 한다. 그것의 상위클라스는 체계클라스인 Applet클라스이다. 바로 이 Applet클라스의 하위클라스를 통해서만 Applet와 열람기의 배합을 완성할수 있다. 아래의 명령문은 전형적인 Applet프로그람의 한 부분이다.

```
import java.applet.Applet;
public class MyApplet extends Applet
{
    …
}
```

1) Applet클라스

Applet클라스는 Java클라스서고에서 중요한 체계클라스로서 java.applet패키지에 존재한다. 클라스계승구성으로부터 보면 Applet클라스는 도형사용자대면부를 만드는 java.awt패키지에 속해야 하지만 실제상 특수한것이므로 체계에서 전문으로 그에 대한 패키지를 만들고있다. Applet클라스는 또한 Java의 체계클라스 java.awt.Panel의 하위클라스이며 Panel은 Container의 일종이다. Applet클라스는 아래와 같은 작용을 한다.

- 다른 대면부요소(례: 단추, 대화칸 등)를 받아들이고 배렬한다.
- 그것이 범위내에 받아들인 사건에 응답하거나 사건을 상위계층으로 전달한다.

Applet는 열람기나 Applet생명주기와 련관한 전문메쏘드도 가지고있다.

2) Applet클라스의 주요메쏘드

사용자가 정의한 Applet하위클라스는 Java Applet프로그람의 명칭이다. 실제적인 실행과정에서 열람기는 바이트코드를 내리적재하는 동시에 자동적으로 사용자 Applet하위클라스의 객체를 창조할수 있으며 적당한 사건발생시 이 객체의 몇가지 주요메쏘드를 자동호출한다.

- init( )메쏘드

init( )메쏘드는 주클라스객체의 초기화작업을 완성하는데 리용된다. Applet의 바이트코드파일이 WWW봉사기측으로부터 내리적재된 후에 열람기는 Applet클라스의 객체를 창조하고 Applet클라스로부터 계승한 init( )메쏘드를 호출한다. 사용자프로그람은 상위클라스의 init( )메쏘드를 재정의하여 일련의 초기화조작(례: 프로그람실행이 요구하는 객체실례를 창조하고 초기화)을 정의할수 있으며 도형이나 문자를 기억기에 적재하고 각종 파라메터를 설정하며 도형과 음성 등을 적재할수 있다.

- start( )메쏘드

start( )메쏘드는 열람기를 기동하여 Applet의 주토막처리를 실행하는데 리용된다. 열람기는 init( )메쏘드를 호출하여 Applet클라스의 객체를 초기화한 다음 start( )메쏘드를 자동호출하여 이 객체를 실행하는 주흐름을 기동한다. 사용자프로그람은 Applet

클라스의 start( )메쏘드를 재정의하여 객체가 활성화되는 경우 실현하려는 련관기능(례: 동화상을 기동하고 파라메터전달을 완성하는 등)을 추가할수 있다.

start( )메쏘드는 또한 Applet가 다시 기동할 때 체계에 의해 자동호출될수 있다. 일반적으로 Applet의 반복기동에는 2가지 경우가 있다. 하나는 사용자가 열람기를 적재할 때이고 다른 하나는 사용자가 열람기를 다른 HTML페지에로 이행하였다가 다시 되돌아오는 경우이다. 총적으로 Applet를 포함하는 HTML페지가 다시 적재될 때 start()메쏘드를 다시 호출할수 있으나 init()메쏘드는 한번만 호출할수 있다.

· paint()메쏘드

paint( )메쏘드의 주요기능은 Applet의 대면부에 문자, 도형과 다른 대면부요소를 현시하는것이다. 이것 역시 열람기가 자동호출할수 있는 Applet클라스의 메쏘드이다.

Applet는 기동후 자동적으로 paint()를 호출하여 자기의 대면부를 그린다.

Applet가 있는 열람기는 창문이 변경될 때 즉 창문의 확대, 축소, 이동, 체계의 다른 부분에 의한 가리우기 등의 경우에 대면부를 반복그려야 하는데 이때 paint( )메쏘드를 자동호출하여 이 작업을 완성한다.

Applet에는 paint( )메쏘드와 련관이 있는 repaint( )메쏘드가 있다. 이 메쏘드가 호출되면 체계는 먼저 update( )메쏘드를 호출하여 Applet객체가 차지하고있는 화면공간을 지운 다음에 paint()메쏘드를 호출하여 다시 그린다.

paint( )메쏘드는 한개의 고정파라메터인 Graphics클라스의 객체 g를 가진다. Graphics클라스는 비교적 단순한 도형사용자대면부작성을 완성하는데 리용된다. 여기에는 원, 점, 선, 다각형을 그리고 간단한 본문을 현시하는 메쏘드들이 포함된다. Applet클라스객체가 초기화되여 기동될 때 열람기는 Graphics클라스의 객체 g를 자동생성하며 g를 파라메터로 하여 Applet클라스객체의 paint( )메쏘드에 전달한다. 이때 paint( )메쏘드에서 객체 g의 메쏘드들을 리용하여 Applet의 대면부를 그려낼수 있다. 프로그람에서 체계가 정의하는 paint( )메쏘드를 재정의하면 Applet대면부를 리용하여 예정화면을 현시할수 있다. 아래에서는 Applet대면부에서 자리표 (0,0)으로부터 (100,100)까지 직선을 그려내는 실례를 보여준다.

실례 5-1

```
Example 5-1 MyApplet_paint.java
1: import java.applet.Applet ;
2: import java.awt.Graphics ;
3: public class MyApplet_paint extends Applet
4: {
5:      public void paint( Graphics g )
6:      {
7:           g.drawLine( 0, 0, 100, 100 );
8:      }
9: }
```

· stop( )메쏘드

stop( )메쏘드는 start( )메쏘드의 거꿀조작과 류사하다. 사용자가 다른 WWW페지를 열람하거나 다른 체계응용으로 절환할 때 열람기는 Applet클라스의 stop메쏘드를 자동호출한다. 프로그람에서 Applet클라스의 stop( )메쏘드를 재정의할수 있으며 또한 일련의 필요한 조작(례: Applet의 동화상조작중지 등)을 재정의할수 있다.

· destroy( )메쏘드

사용자가 열람기에서 탈퇴할 때 열람기에서 실행하는 Applet실체 역시 없어진다. 즉 기억기해제된다. Applet가 없어지기전에 열람기는 Applet실체의 destroy( )메쏘드를 자동호출하여 자원해방, 접속닫기의 클라스조작을 완성한다.

Applet실체자체는 열람기에 의하여 창조되고 열람기에 의하여 삭제되므로 destroy( )메쏘드에서 특별히 정의하지 않아도 된다. 실제상 우에서 서술한 Applet가 열람기로부터 자동호출하는 주요메쏘드인 init( ), start( ), stop( ), destroy( )는 각각 Applet의 초기화, 기동, 림시정지, 소멸까지의 생명주기의 매개 단계에 대응한다. 그림 5-1은 이 관계를 보여주고있다.

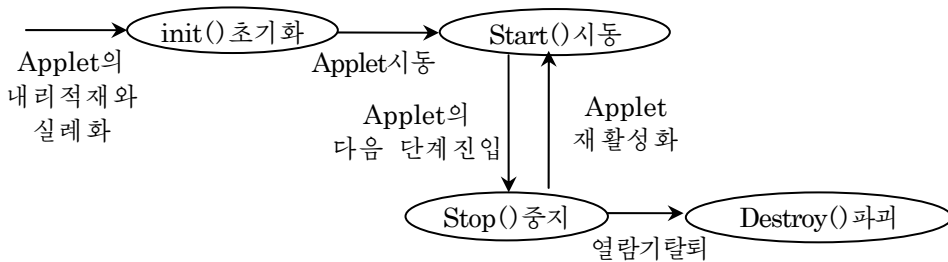



그림 5-1. Applet의 생명주기와 주요메쏘드

실례 5-2에서는 우에서 서술한 몇개의 메쏘드를 리용하고있으며 실행후에 이 Applet주요메쏘드들의 집행시각과 Applet생명주기와의 관계를 현시한다.

실례 5-2

```
Exaple 5-2 LifeCycle.java:
1: import java.applet. * ;
2: import java.awt. * ;
3: public class LifeCycle extends Applet
4: {    // 매 계수기의 정의
5:      private int InitCnt;
6:      private int StartCnt;
7:      private int StopCnt;
8:      private int DestroyCnt;
9:      private int PaintCnt;
10:     public LifeCycle( )
```

```
11:     {  //매 계수기들의 초기화
12:         InitCnt = 0; StartCnt = 0; StopCnt = 0; DestroyCnt = 0; PaintCnt = 0;
13:     }
14:     public void init( )
15:     {
16:         InitCnt++ ;
17:     }
18:     public void destroy( )
19:     {
20:         DestroyCnt++ ;
21:     }
22:     public void start( )
23:     {
24:         StartCnt++ ;
25:     }
26:     public void stop( )
27:     {
28:         StopCnt++ ;
29:     }
30:     public void paint(Graphics g)
31:     {
32:         PaintCnt++ ;
33:         g.drawLine(20, 200, 300, 200); g.drawLine(20, 200, 20, 20);
34:         g.drawLine(20, 170, 15, 170); g.drawLine(20, 140, 15, 140);
35:         g.drawLine(20, 110, 15, 110);
36:         g.drawLine(20, 80, 15, 80);
37:         g.drawLine(20, 50, 15, 50);
38:         g.drawString("Init( )", 25, 213);
39:         g.drawString("Start( )", 75, 213);
40:         g.drawString("Stop( )", 125, 213);
41:         g.drawString("Destroy( )", 175, 213);
42:         g.drawString("paint( )", 235, 213);
43:         g.fillRect(25, 200-InitCnt * 30, 40, InitCnt * 30);
44:         g.fillRect(75, 200-StartCnt * 30, 40, StartCnt * 30);
45:         g.fillRect(125, 200-StopCnt * 30, 40, StopCnt * 30);
46:         g.fillRect(175, 200-DestroyCnt * 30, 40, DestroyCnt * 30);
47:         g.fillRect(235, 200-PaintCnt * 30, 40, PaintCnt * 30);
48:     }
49:}
```

프로그람설명

이 프로그람은 Applet의 5개 주요메쏘드의 집행회수를 계산하여 막대도표로 표시한다. 프로그람에서는 Graphics클라스의 몇가지 메쏘드(례 : drawLine( ), drawstring( ), fillRect( ))를 사용하고있다. 그림 5-2는 실례 5-2의 어떤 시각에서의 집행결과이다.

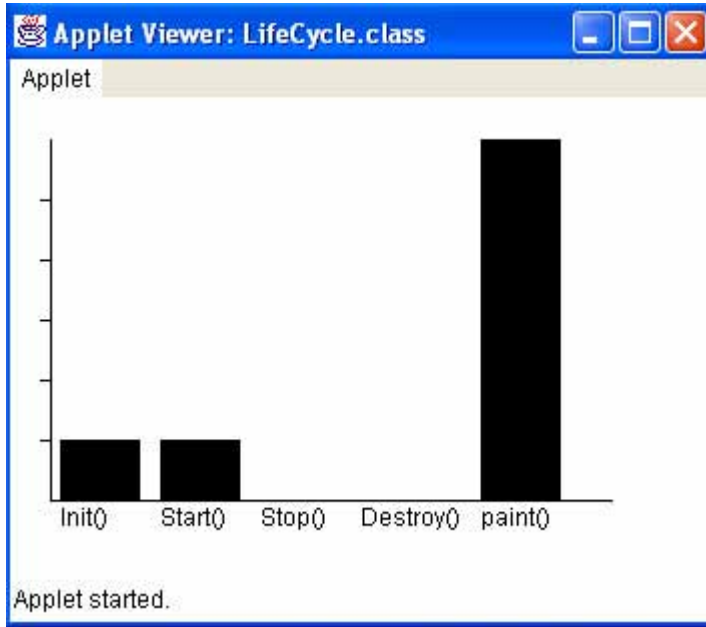



그림 5-2. 실례 5-2의 실행결과

5.2.3. HTML파일에서 파라메터전달

Applet는 HTML파일에 삽입하여 열람기에서 실행하는 프로그람이다. 프로그람작성단계는 원천코드로부터 바이트코드를 번역하는 프로그람작성단계까지는 Java Application과 차이가 거의 없다. 그러나 Applet를 리용하려면 반드시 HTML문서와 서로 배합하여야 한다. 앞에서 이미 소개한것처럼 HTML은 하이퍼본문꼬리표언어로서 여러가지 꼬리표를 통하여 하이퍼본문정보를 편집배렬한다. HTML문서에서 Applet삽입은 약속한 특수꼬리표를 똑같이 사용할것을 요구한다. 실례로 <APPLET>와 </APPLET>는 Applet를 삽입하는 꼬리표으로서 여기에 적어도 3개의 파라메터 code, height, width를 포함하여야 한다. <APPLET>꼬리표에서 선택가능한 다른 파라메터를 사용할수 있다.

codebase: Applet바이트코드파일의 보존위치가 그것을 삽입한 HTML문서와 같지 않을 때 파라메터 codebase를 사용하여 바이트코드파일의 위치를 지적해야 한다. 이때 이 위치는 URL의 형식을 가져야 한다. 즉

codebase = http://www.illusion,org/Applet/MyApplet_paint.class

alt: 만일 HTML페지를 해석하는것이 Java해석기를 포함하지 않는 열람기이면 그것은 바이트코드를 집행할수 없으며 alt파라메터가 지정하는 정보를 사용자에게 현시한다. 즉

alt = "This a Java Applet your browser can not understand."

align: 열람기창문에서 Applet대면부의 위치를 표시한다.

align = CENTER

HTML파일은 삽입된 Applet에 파라메터를 전달할수 있으므로 Applet의 실행을 더 잘할수 있다. 이것은 HTML파일의 꼬리표 <PARAM>을 통하여 완성한다.

```
<HTML>
  <BODY>
    <APPLET code = "MyApplet_param.class" height = 200 width = 300>
      <PARAM name = vstring value = "이것은 HTML에서 온 파라메터입니다.">
      <PARAM name = x value = 50>
      <PARAM name = y value = 100>
    </APPLET>
  </BODY>
</HTML>
```

이 HTML파일에서는 이름이 MyApplet_param인 Applet를 삽입하고있으며 동시에 Applet실행시 3개의 문자렬파라메터를 전달한다. 하나는 파라메터이름이 vstring으로서 취하는 값은 "이것은 HTML에서 온 파라메터이다"이며 다른 두개는 파라메터이름이 각각 x와 y로서 취하는 값은 "50"과 "100"이다. 보는바와 같이 매개 <PARAM>꼬리표는 오직 하나의 문자렬형의 파라메터만을 전달할수 있다. 파라메터이름은 그것을 다른 파라메터와 구분하는데 쓰이며 name을 리용하여 지정한다. 이 파라메터의 값은 value를 리용하여 지정한다. 아래에서 MyApplet_param.java가 어떻게 HTML파일로부터 파라메터를 얻는가를 고찰하여보자.

실례 5-3

```
Example 5-3 MyApplet_param.java
1: import java.applet.Applet ;
2: import java.awt.Graphics ;
3: public class MyApplet_param extends Applet
4: {
5:      private String s = "" ; //HTML파라메터를 접수하는데 쓰이는 변수
6:      private int x ;
7:      private int y ;
8:      public void init( )
```

```
9:     {
10:        s = getParameter ( "vstring" ) ; //HMTL에서 전달되는 파라메터접수
11:        x = Integer.parseInt ( getParameter ("x"));
12:        y = Integer.parseInt ( getParameter ("y"));
13:    }
14:    public void paint ( Graphics g )
15:    {
16:        if(s != null)
17:            g.drawString (s, x, y);
18:    }
19:}
```

프로그람설명

실례 5-3의 실행결과를 그림 5-3에서 보여준다.

Applet는 getparameter( )메쏘드를 리용하여 HTML이 전달하는 파라메터를 얻는다. 이 메쏘드는 문자렬파라메터를 가지고 얻으려는 HTML파라메터의 이름(즉 name이 지정한 파라메터)을 나타낸다. 메쏘드의 귀환값은 한개의 문자렬객체이다. 즉 HTML파일에서 value가 지정하는 문자렬이다. 만일 이 파라메터가 다른 류형으로 되자면 Application이 지령행파라메터와 같이 파라메터형변환을 하여야 한다.

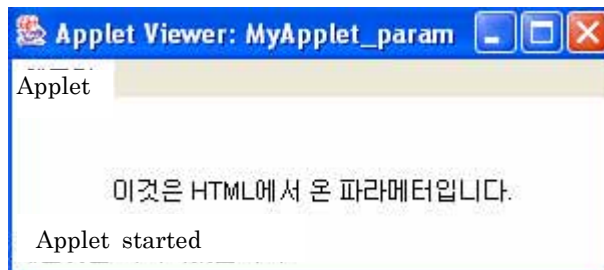



그림 5-3. 실례 5-3의 실행결과

제6장. 례외처리와 다중토막처리

제1절. 례외처리

- 례외던지기의 형식
  장식부 귀환형 메쏘드이름(파라메터목록) throws 례외클라스이름목록
  {…
  throw 례외클라스이름; … }
- 례외처리의 형식
  try … catch … finally

6.1.1. 례외와 례외클라스

례외(Exception)는 Java언어에만 있는 특정한 실행오유처리기구이다. Java프로그람은 망환경에서 실행하므로 안전성을 보장하는것이 우선적으로 고려해야 할 중요한 것으로 되고있다. 프로그람의 실행오유를 즉시 유효하게 처리하기 위하여 Java에서는 례외와 례외클라스를 도입하였다.

Java에서는 많은 례외클라스를 정의하고있다. 매개 례외클라스는 일종의 실행오유를 나타내며 클라스에는 이 실행오유의 정보와 오유처리메쏘드 등의 내용이 포함된다. Java프로그람실행과정에서 식별할수 있는 실행오유가 발생할 때 즉 오유가 례외클라스에 대응할 때 체계는 상응한 이 례외클라스의 객체를 생성할수 있다. 즉 례외가 발생된다. 일단 례외객체가 생겼으면 체계에서는 반드시 상응한 기구를 가지고 그것을 처리하여 전체프로그람실행의 안전성을 보장하여야 한다. 이것이 바로 Java의 례외처리기구이다.

1) 례외클라스의 구조와 구성

Java의 례외클라스는 실행시 오유를 처리하는 특수한 클라스이며 매 례외클라스는 특정한 실행오유에 대응한다. Java례외클라스들은 모두 Exception클라스의 하위클라스이다. 이 클라스계승구조를 그림 6-1에서 보여준다.

Throwable클라스는 클라스서고 java.lang패키지의 클라스이며 2개의 하위클라스 즉 Exception와 Error를 파생하고있다. 여기서 Error클라스는 체계준위의 클라스이며 Exception클라스는 응용프로그람에서 주로 사용하는 클라스이다.

다른 클라스와 마찬가지로 Exception클라스도 자체의 메쏘드와 속성, 2개의 구성자를 가진다.

```
public Exception( );
public Exception(String s);
```

두번째 구성자에서 문자렬파라메터는 전달하려는 정보를 접수하는데 이 정보는 보통 이 례외에 대응하는 오유에 대한 서술이다.

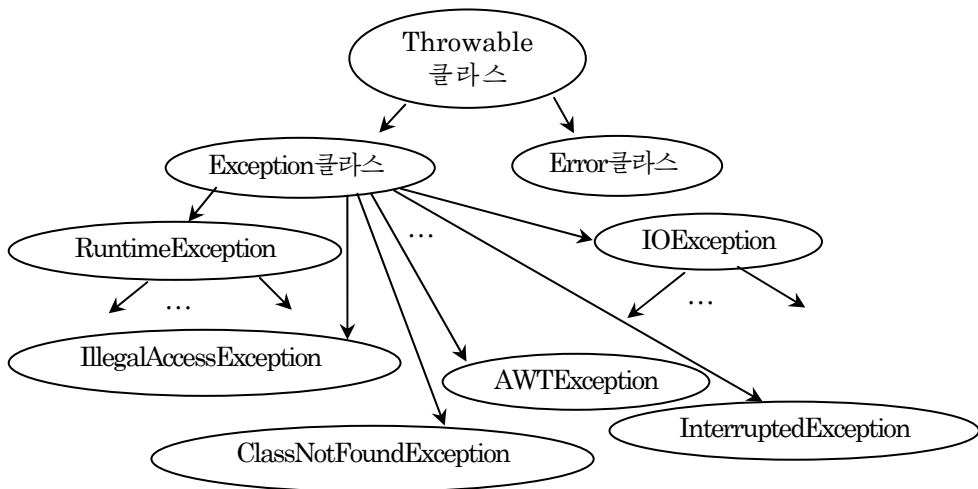



그림 6-1. Exception클라스의 계승구조

Exception클라스는 상위클라스 Throwable로부터 몇개의 메쏘드를 계승하고있다. 여기서 자주 쓰이는것은 toString메쏘드이다.

public String toString( )

이 메쏘드는 현재 Exception클라스정보를 표현하는 문자렬을 귀환시킨다.

2) 체계정의의 실행례외

Exception클라스는 몇개의 하위클라스를 가지며 매개 하위클라스는 실행시의 구체적인 오유를 나타낸다. 이 하위클라스들은 체계에 미리 정의되여있으며 Java클라스서고에 포함되여있다. 이것을 체계가 정의한 **실행례외**라고 한다.

체계가 정의한 실행례외는 보통 체계실행오유에 대응한다. 이 오유는 조작체계오유뿐아니라 전체 체계의 마비를 초래할수 있기때문에 례외클라스를 정의하여 특별한 처리를 하여야 한다.

표 6-1에서는 자주 보게 되는 체계정의의 례외를 보여준다. 표에서 보는것처럼 상응한 례외들을 정의하였기때문에 Java프로그람이 오유(례: 빈 객체의 인용 등)를 발생시키면 체계는 대응하는 례외객체를 자동적으로 생성하여 이 오유를 처리조종할수 있다.

표 6-1. 체계정의의 실행례외

체계정의의 실행례외	례외에 대응하는 체계실행오유
ClassNotFoundException	내장하려는 클라스를 찾지 못했습니다.
ArrayIndexOutOfBoundsException	배렬한계넘침사용
FileNotFoundException	지정한 파일이나 등록부를 찾지 못했습니다.
IOException	입출력오유
NullPointerException	빈 기억공간이 없는 객체를 인용
ArithmeticException	산수오유 례: 나눔수가 령인 경우
InterruptedException	토막처리가 갑자기 기다리거나 기타 원인에 의한 림시정지시 다른 토막처리에 의해 단절되는 경우
UnknownHostException	주콤퓨터의 IP주소를 확정할 메쏘드가 없는 경우
SecurityException	안전성오유 례: Applet가 파일을 읽기 쓰기하려는 경우
MalformedURLException	URL격식오유

3) 사용자정의의 례외

체계가 정의하는 례외는 주로 체계가 예견할수 있는 실행오유를 처리하는데 쓰이며 응용이 특수한 실행오유에 대하여서는 프로그람작성자가 프로그람의 특수한 론리에 따라 프로그람에서 례외클라스와 례외객체를 창조하여야 한다.

실례로 아래와 같은 《대기렬빠져나오기》메쏘드를 정의하려고 한다고 하자.

```
int dequeue( ) //대기렬빠지기조작, 대기렬이 비지 않았으면 대기렬머리부로
                부터 한개의 자료를 추출
{
    int data;
    if(!isEmpty( ))
    {
        data = m_FirstNode.getData( );
        m_FirstNode = m_FirstNode.getNext( );
        return data;
    }
    else
        return -1;
}
```

이 메쏘드에서 대기렬이 이미 비여있으면 《대기렬빠져나오기》는 -1을 귀환시켜 《대기렬빠져나오기》조작실패를 표시할수 있다. 이러한 처리는 매우 불편하다. 그것은 대기렬에서 -1을 보존하고있어야 하며 동시에 dequeue( )를 호출하는 다른 메쏘드

들이 이 오유발생의 규약을 알고있어야 하기때문이다. 이 문제를 해결하기 위하여 사용자프로그람의 례외 EmptyQueueException을 정의하여 우의 《빈 대기렬로부터 빠져나오기》의 론리적오유를 전문적으로 처리할수 있다.

```
class EmptyQueueException extends Exception
    // 사용자가 정의한 체계클라스의 하위클라스
{
    Queue sourceQueue;
    public EmptyQueueException(Queue q)
    {
        super(".");
        sourceQueue=q;
    }
    public String toString( )
      // 상위클라스의 메쏘드를 다중정의, 상세한 오유정보를 준다.
    {
        return("대기렬은" + sourceQueue.toString() + "이미 비였음";
    }
}
```

사용자정의의 례외는 프로그람에서 발생할수 있는 론리적오유를 처리하는데 쓰이며 이 오유를 체계에 의하여 즉시 식별처리하게 한다. 따라서 사용자프로그람이 보다 좋은 오유견딤성능(fault-tolerant)을 가지고 전체 체계를 보다 안전하게 한다.

사용자정의의 례외를 창조할 때 일반적으로 아래의 단계를 거쳐야 한다.

· 새로운 례외클라스를 선언한다. Exception클라스나 이미 존재하는 다른 체계례외클라스 또는 사용자례외클라스는 상위클라스로 된다.

· 새로운 례외클라스에 대하여 속성과 메쏘드를 정의하거나 상위클라스의 속성과 메쏘드를 재정의하여 이 속성과 메쏘드들이 클라스에 대응하는 오유정보를 능히 표현할수 있게 한다.

례외클라스를 정의하면 체계는 특정한 실행오유를 능히 식별할수 있으며 실행오유를 제때에 조종처리할수 있다.

6.1.2. 례외던지기

Java프로그람은 실행시 식별할수 있는 오유를 일으키면 이 오유에 대응하는 례외클라스의 객체를 발생시키며 이 과정을 **례외던지기**라고 한다. 례외클라스가 다르면 례외를 던지는 메쏘드도 같지 않다.

1) 체계가 자동적으로 진행하는 례외던지기

체계가 정의하는 모든 실행례외는 체계에 의하여 자동적으로 던져질수 있다.

실례 6-1은 나누는 수가 령으로 될 때 나타나는 연산례외를 시험검사한다. 이 실례를 통하여 체계가 정의하는 실행례외를 어떻게 사용하는가를 알수 있다.

실례 6-1

```
Example 6-1 TestSystemException.java
1: public class TestSystemException
2: {
3:      public static void main(String args[])
4:      {
5:          int a = 0, b = 5;
6:          System.out.println(b / a); //령으로의 나누기, 체계정의의 례외가 발생
7:      }
8: }
```

프로그람설명

우의 프로그람은 간단한 Java Application으로서 오유는 나누는 수가 령으로 되는 경우이며 실행과정에 산수례외(ArithmeticException)를 일으킨다. 이 례외는 체계가 미리 정의한 클라스이며 이러한 오유에 부딪치면 프로그람의 실행흐름을 자동중지하고 ArithmeticException클라스의 객체를 새로 창조한다. 즉 산수연산례외를 던진다. 그림 6-2는 실례 6-1의 실행결과이다.

```
D:\Javatextbook\Test>javac TestSystemException.java

D:\Javatextbook\Test>java TestSystemException
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestSystemException.main(TestSystemException.java:7)

D:\Javatextbook\Test>
```

그림 6-2. 실례 6-1의 실행결과

2) 명령문에 의해 진행하는 례외던지기

사용자프로그람이 정의하는 례외는 체계에 의하여 자동적으로 던져질수 없으며 반드시 throw명령문에 의하여 어느것이 이 례외에 대응하는 오유를 발생시킨것인가를 정의하고 이 례외클라스의 새로운 객체를 던진다. throw명령문을 리용하여 례외객체를 던지는 문법형식은 다음과 같다.

```
장식부 귀환형 메쏘드이름(파라메터목록) throws 례외클라스이름목록
 {
   ......
   throw 례외클라스이름;
   ......
 }
```

throw명령문을 사용하여 례외상황을 전달할 때 아래의 문제에 주의해야 한다.

일반적으로 례외를 던지는 명령문은 일정한 조건을 만족시킬 때 집행하는것으로 정의한다. 실례로 throw명령문을 if명령문의 if분기에 놓고 오직 if조건이 만족할 때 즉 사용자가 정의한 론리적오유의 발생시에야 집행한다.

throw명령문을 포함하는 메쏘드는 메쏘드머리부정의에서 아래의 부분을 포함시켜야 한다.

```
throws 례외클라스이름목록
```

이렇게 하여 이 메쏘드를 사용하려는 모든 메쏘드를 통지한다.

이 메쏘드는 throw명령문을 포함하므로 그것을 접수처리하여 실행과정에서 던질수 있는 례외를 준비해야 한다. 만일 메쏘드에서의 throw명령문이 하나가 아니고 메쏘드머리부의 례외클라스이름목록 역시 하나가 아니면 발생할수 있는 모든 례외를 포함하여야 한다.

실례로 Queue클라스의 dequeue( )메쏘드는 EmptyQueueException례외객체를 던질수 있다.

```
int dequeue( ) throws EmptyQueueException // 대기렬빠지기조작, 대기렬
                이 비지 않으면 대기렬머리부로부터 한개의 자료를 추출
{
    int data;
    if(isEmpty( ))
        throw(new EmptyQueueException(this));
    else
    {
        data = m_FirstNode.getData( );
        m_FirstNode = m_FirstNode.getNext( );
        return data;
    }
}
```

우의 프로그람에서 알수 있는것처럼 체계는 사용자가 정의한 례외를 식별하고 창조할수 없으므로 작성자가 프로그람의 적당한 위치에서 례외를 자체정의하는 객체를 창조하고 throw명령문을 리용하여 이 새로운 례외객체를 던져야 한다.

6.1.3. 례외처리

례외처리는 주로 례외잡기, 프로그람흐름의 뛰여넘기, 례외처리명령문블로크의 정의를 포함하고있다.

1) 례외잡기

례외가 던져질 때 전문적인 명령문을 가지고 던져지는 객체를 접수하여야 한다. 이 과정을 **례외잡기**라고 한다. 례외클라스의 객체가 포착되거나 접수된 후 체계는 현재의 흐름을 중지하고 전문적인 례외처리의 명령문블로크에 뛰여넘을수 있다. 혹은 직접 현재의 프로그람과 Java가상기계를 뛰여넘어 조작체계에로 돌아올수 있다.

Java프로그람에서 례외객체는 례외처리명령문블로크에 의하여 처리한다. 례외처리명령문블로크를 catch명령문블로크라고 하며 문법형식은 다음과 같다.

```
catch(례외클라스이름 례외형식파라메터이름)
 {
   례외처리명령문들;
 }
```

Java에서 매개 catch명령문블로크는 try명령문블로크와 쌍으로 리용하여야 한다. try명령문블로크는 Java의 례외처리기구를 기동하는데 쓰이며 례외를 전달하는 명령문인 throw명령문과 례외메쏘드를 호출하는 메쏘드호출명령문들을 포함한다.(실례 6-2)

실례 6-2

```
Example 6-2 TestQueueException.java
1: public class TestQueueException
2: {
3:      public static void main(String args[])
4:      {
5:           Queue queue = new Queue( );
6:
7:           for(int i = 1; i < 8; i++)
8:           {
9:                queue.enqueue(i);
10:               System.out.println(queue.visitAllNode( ));
11:          }
12:          System.out.println("\n"); //두행 되돌이의 추가
13:          try{
14:               while(! queue.isEmpty( ))
15:               {
```

```
16:                 System.out.print(queue.dequeue( ) + "대기렬에서 빠져나오기;");
17:                 System.out.println("대기렬에 남아있는것:" + queue.visitAllNode( ));
18:             }
19:         }
20:         catch(EmptyQueueException e)
21:         {
22:             System.out.println(e.toString( ));
23:         }
24:     }
25:}
```

프로그람설명

실례 6-2의 프로그람은 클라스 Node, LinkList와 Queue의 정의를 생략하였다. 13-18행에서 정의하는 try블로크에서는 EmptyQueueException객체를 던질수 있는 dequeue( )메쏘드를 호출하였고 catch블로크는 전문적으로 이 례외를 잡기하는데 쓰인다. 보는바와 같이 catch명령문블로크는 try명령문블로크의 뒤에 놓이면서 함께 쓰이고있다. try명령문블로크에서 어떤 명령문의 집행시 례외를 발생시키게 될 때 이때 기동되는 례외처리기구는 그것을 자동잡기할수 있다. 그다음 흐름은 자동적으로 례외를 발생시키는 명령문뒤의 집행하지 않은 명령문들을 뛰여넘어 catch명령문블로크에로 이행하여 catch블로크의 명령문들을 집행한다.

2) 다중례외처리

catch블로크는 try블로크의 뒤에 있으면서 try블로크가 발생시킬수 있는 례외들을 접수하는데 쓰인다. catch명령문블로크는 보통 같은 방식을 리용하여 접수한 모든 례외들을 처리할수 있다. 그러나 실제상 한개의 try블로크는 여러 종류의 서로 다른 례외들을 발생시킬수 있으며 만일 서로 다른 메쏘드를 취하여 이 례외들을 처리하려면 다중례외처리기구를 사용하여야 한다.

다중례외처리는 try블로크뒤에서 여러개의 catch블로크들을 정의하여 실현하며 이때 매개 catch블로크는 특정한 례외객체를 접수처리하는데 쓰인다. 서로 다른 catch블로크를 리용하여 각각 서로 다른 례외객체를 처리하려면 우선 catch블로크가 서로 다른 례외객체들을 구별하여야 하며 례외객체를 이 블로크에서 접수처리해야 하는가를 판단해야 한다. 이 판단은 catch블로크의 파라메터를 통하여 실현된다.

그러므로 try블로크뒤에는 가능한 몇개의 catch블로크가 있을수 있으며 매개 catch블로크의 파라메터는 하나의 례외클라스이름으로 된다. try블로크가 례외를 던질때 프로그람의 흐름은 우선 첫번째 catch블로크로 이행하며 례외객체를 이 catch블로크에서 접수할수 있는가를 판단한다. 례외객체가 catch명령문블로크에 의해 접수될수

있는가 없는가는 이 례외객체와 catch블로크와의 례외파라메터의 정합상태를 고찰하여야 한다. 3가지 조건중 임의의 1개를 만족시킬 때 례외객체는 접수된다.

- 례외객체는 파라메터와 같은 례외클라스에 속한다.
- 례외객체는 파라메터례외클라스의 하위클라스에 속한다.
- 례외객체는 파라메터가 정의하는 대면을 실현한다.

만일 try블로크가 발생시킨 례외객체가 첫번째 catch블로크에 의해 접수된다면 프로그람의 흐름은 직접 이 catch명령문블로크에로 뛰여넘으며 명령문블로크집행을 완성한다. 즉 현재의 메쏘드에서 탈퇴하여 try블로크에서 집행하지 못한 명령문과 다른 catch블로크는 무시한다. 만일 try블로크가 발생시키는 례외객체가 첫번째 catch블로크와 정합하지 않으면 체계는 두번째 catch블로크에로 자동적으로 이동하여 정합을 하며 이렇게 하여 이 례외객체를 접수할수 있는 catch블로크를 찾는다. 즉 흐름의 뛰여넘기를 진행한다.

만일 모든 catch블로크들이 현재의 례외객체와 정합할수 없으면 메쏘드는 이 례외객체를 처리할수 없으며 프로그람흐름은 이 메쏘드를 호출하는 상층메쏘드로 귀환된다. 만일 이 상층메쏘드에서 발생하는 례외객체와 정합하는 catch블로크가 있다면 흐름은 이 catch블로크에로 이행하며 그렇지 않으면 상층의 메쏘드를 계속 추적한다. 만일 모든 메쏘드들에서 적당한 catch블로크를 찾을수 없으면 Java체계가 자체로 이 례외객체를 처리한다. 이때 프로그람의 집행을 중지할수 있으며 가상기계에서 탈퇴되여 조작체계에로 돌아와 표준출력에 필요한 례외정보를 인쇄한다.

catch블로크를 작성하여 서로 다른 례외를 처리할 때 일반적으로 다음의 문제에 주의하여야 한다. catch블로크에서의 명령문은 례외의 불일치에 따라 서로 다른 조작을 집행하여야 한다. 보통 이 조작에는 례외와 오유에 대한 정보를 현시하고 례외이름과 례외를 일으키는 메쏘드이름 등이 포함된다.

례외객체와 catch블로크의 정합은 catch블로크의 선후차에 따라 집행하므로 다중례외를 처리할 때 매 catch블로크의 해결순서에 주의하여 심중히 작성하여야 한다. 일반적으로 비교적 구체적이고 자주 보는 례외에 대한 catch블로크처리는 앞에 놓아야 하며 다중례외와 서로 정합할수 있는 catch블로크는 될수록 뒤에 놓아야 한다.

제2절. Java다중토막처리기구

- 토막처리는 처리보다 작은 집행단위이다.
- 토막처리의 생명주기: 창조 - 준비완료 - 실행 - 막기 - 소멸
- 토막처리의 창조메쏘드 - Runnable대면의 실현

앞에서 개발한 프로그람들은 대부분 단일토막처리이다. 즉 프로그람은 머리부로부터 마지막까지 하나의 집행경로만을 가진다. 그러나 현실세계의 많은 과정의 실현은 여러 과정을 갖추고 동시에 동작한다.(례: 봉사기가 가능한 여러대의 의뢰기의 요구를 동시에 처리하는것 등)

다중토막처리는 여러개의 집행객체가 동시에 존재하고 여러개의 서로 다른 집행경로를 따라 함께 작업하는 경우를 말한다. Java언어의 중요한 특징은 바로 다중토막처리기능을 편리하게 리용할수 있게 하고 여러 과제를 동시에 처리할수 있는 응용프로그람을 만들수 있다는데 있다.

6.2.1. Java에서의 토막처리

1) 프로그람처리토막처리

프로그람은 정적코드이며 응용쏘프트웨어가 집행하는 원본이다.

처리는 프로그람의 한번의 동작집행과정으로서 코드로부터 적재, 집행완성까지의 전과정에 대응한다. 이 과정 역시 처리자체가 발생, 발전하여 넘어갈 때까지의 전과정이다. 집행원본으로 되는 동일한 프로그람은 체계에 적재되는 서로 다른 기억기구역에서 각각 집행될수 있으며 서로 다른 처리과정을 형성한다.

토막처리는 처리(Process)보다 작은 집행단위이다. 처리는 그 집행과정에서 여러개의 토막처리를 생성할수 있으며 여러 집행토막처리를 형성한다. 매 토막처리도 발생, 존재, 소멸과정을 가지는 하나의 동적인 개념으로 되고있다. 매개 처리는 전용기억기구역을 가진다.

2) 토막처리의 상태와 생명주기

Java프로그람은 기정인 주토막처리를 가진다. Application의 주토막처리는 main( )메쏘드가 집행하는 토막처리이고 Applet의 주토막처리는 열람기를 통하여 Java프로그람을 적재하고 집행하는것을 말한다. 다중토막처리를 실현하려면 반드시 주토막처리에서 새로운 토막처리객체를 창조하여야 한다. Java언어는 Thread클라스와 그의 하위클라스의 객체를 사용하여 토막처리를 표시하며 새로 창조하는 토막처리의 전체 생명주기는 아래의 5가지 상태를 갖추어야 한다.

(1) 창조

Thread클라스나 그의 하위클라스의 객체가 선언되고 창조될 때 새로 생기는 토막처리객체는 새로 만들어지는 상태에 놓인다. 이때 그것은 상응한 기억기공간과 다른 자원들을 가지고있으며 이미 초기화된다.

(2) 준비완료

새로 만들어지는 상태에 있는 토막처리는 기동된 후 토막처리대기렬에 들어가며 CPU시간을 기다린다. 이때 그것은 이미 실행조건을 갖추고있다. 일단 자기의 차례가 되여 CPU자원을 공유할 때 그것을 창조한 주토막처리에서 탈퇴하여 자기의 생명주기를 독립적으로 개시한다. 그리고 원래 막기상태에 있던 토막처리가 막기해제된 후에 준비완료상태에 들어간다.

(3) 실행

준비완료상태의 토막처리가 처리되여 처리기자원을 얻을 때 실행상태에 들어간다. 매개 Thread클라스와 그의 하위클라스의 객체는 run( )메쏘드를 가지며 토막처리객체가 배분집행될 때 이 객체의 run( )메쏘드를 자동호출하여 첫째 행부터 순차적으로 집행한다. run()메쏘드는 이 토막처리의 조작과 기능을 정의하고있다.

(4) 막기

현재 집행하고있는 토막처리는 어느 특수한 정황(례: 입출력작성시)에서 CPU에 의해 잠간 집행이 중지되며 막기상태에 들어가게 된다. 막기상태시 그것은 대기렬에 들어갈수 없다. 단지 막기를 일으키는 원인이 제거될 때에만 토막처리는 준비완료상태에 들어갈수 있으며 다시 토막처리대렬에 진입하여 CPU자원을 기다리다가 차례가 되면 원래 중지위치로부터 시작하여 계속 실행된다.

(5) 소멸

소멸상태에 있는 토막처리는 실행될수 없다. 토막처리가 소멸하는 원인은 두가지이다. 하나는 정상실행하는 토막처리가 그전의 작업을 전부 완성하였기때문이다. 즉 run( )메쏘드의 마지막명령을 집행완료하고 탈퇴하는 경우이다. 다른 하나는 토막처리가 사전에 강제적으로 중지되기때문이다. 례하면 stop( )나 destory( )메쏘드의 집행을 통하여 토막처리를 중지시키는 경우이다.

토막처리와 처리가 같은 점은 하나의 동적인 개념이라는데 있으며 역시 처리와 같이 산생으로부터 소멸까지의 생명주기를 가진다는데 있다.(그림 6-3)

3) 토막처리와 우선권

준비완료상태에 있는 토막처리는 우선 준비완료대기렬에 진입하여 처리기자원을 기다린다. 같은 시각에 준비완료대기렬의 토막처리는 여러개일수 있으며 각자 과제의 준위는 같지 않다. 실례로 화면을 현시하는 토막처리

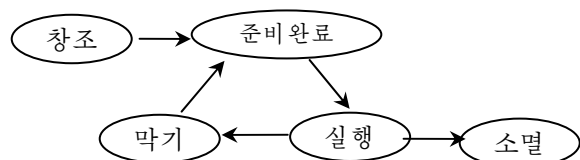



그림 6-3. 토막처리의 생명주기

는 빨리 집행할것을 요구하며 기억기쪼각을 수집하는데 쓰이는 휴지회수토막처리는 급하지 않으므로 처리기가 비여있기를 기다렸다가 다시 집행할수 있다. 이 차이를 고려하여 작업을 보다 합리적으로 조직하기 위해 다중토막처리체계는 매개 토막처리에 한개의 토막처리우선권을 부여할수 있다. 과제가 급하고 중요한 토막처리는 우선권이 높으며 반대의 경우는 우선권이 낮다. 우선권이 높은 토막처리는 앞선위치에 배렬할수 있으며 처리기자원을 우선적으로 리용할수 있다. 한편 우선권이 낮은 토막처리는 그 앞에 배렬한 우선권이 높은 토막처리를 집행완성한 후에야 처리기자원을 얻을수 있다. 우선권이 같은 토막처리에 대하여서는 대기렬의 《선입선출》의 원칙에 따른다. 즉 준비완료상태에 먼저 들어온 토막처리가 처리기자원에 우선적으로 분배되며 뒤이어 대기렬에 들어서는 토막처리가 봉사된다.

6.2.2. Java의 토막처리클라스와 Runable대면

Java에서 다중토막처리의 실현은 2가지 방법으로 한다. 하나는 사용자자체의 토막처리하위클라스를 창조하는것이고 다른 하나는 사용자자체의 클라스에서 Runable대면을 실현하는것이다. 어떤 방법이든지간에 Java기초클라스서고의 Thread클라스와 그의 메쏘드를 사용하여야 한다.

1) Runable대면

Runable대면은 오직 run( )메쏘드만을 가진다. 그러므로 Runable대면을 실현하는 모든 사용자클라스는 반드시 run( )메쏘드를 구체적으로 실현하여야 한다. 즉 메쏘드본체를 작성하고 구체적인 조작을 정의하여야 한다. Runable대면에서의 run( )메쏘드는 비교적 특수한 메쏘드로서 실행체계에 의하여 자동적으로 식별집행된다. 그러므로 Runable대면을 실현한 클라스는 실제상 주토막처리밖의 새로운 토막처리조작을 정의하고있으며 새로운 토막처리조작과 집행흐름의 정의는 다중토막처리응용을 실현하는 가장 중요하고 기본적인 작업의 하나로 된다.

2) Thread클라스

Thread클라스는 토막처리가 가지고있는 속성, 메쏘드외에 다음과 같은것을 가진다.

(1) 구성자

Thread클라스의 구성자에는 여러개가 있으며 대응하는 조작은 다음과 같다.

① public Thread( ): 체계토막처리클라스의 객체를 창조한다.

② public Thread(Runable target): 구성자가 완성되는 조작으로서 Runable대면을 실현하고있는 target객체에서 정의된 run( )메쏘드를 리용하여 새로 창조하려는 토막처리객체의 run( )메쏘드를 초기화하거나 재정의한다.

③ public Thread(String ThreadName): 첫번째 구성자가 작업하는 기초우에서 창조하려는 토막처리객체에 대한 문자렬이름을 지정한다.

④ public Thread(Runable target, String ThreadName): ②, ③의 구성자의 기능을 실현한다.

구성자를 리용하여 새로운 토막처리를 창조한 후에 이 객체의 련관자료를 초기화하고 토막처리생명주기의 첫번째 상태인 창조상태에 들어간다.

(2) 토막처리우선권

Thread클라스에는 토막처리우선권을 표현하는 3개의 정적상수가 있다. 즉 MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY가 있다. 여기서 MIN_PRIORITY는 최저우선권으로서 값은 1이고 MAX_PRIORITY는 최고우선권으로서 값은 10이다. NORM_PRIORITY는 보통 우선권을 나타내며 기정값은 5이다.

새로 창조되는 토막처리는 그것을 창조하는 상위토막처리의 우선권을 계승한다. 상위토막처리는 새로 창조하는 토막처리객체명령문을 집행하는 토막처리를 가리키며 프로그람의 주토막처리일수도 있고 어떤 사용자정의의 토막처리일수도 있다.

일반적으로 주토막처리는 보통의 우선권을 가진다.

사용자는 Thread클라스의 setPriority( )메쏘드를 리용하여 토막처리우선권을 정할수 있다.

(3) 주요메쏘드

① start( )메쏘드: 토막처리객체를 기동하는데 리용되며 새로운 창조상태로부터 준비완료상태로 이전하여 준비완료대기렬에 들어간다.

② run( )메쏘드: Thread클라스의 run( )메쏘드와 Runable대면의 run( )메쏘드의 기능과 작용은 서로 같으며 모두 토막처리객체가 처리된 후에 집행되는 조작을 정의하는데 쓰인다. 또한 체계를 자동호출하고 사용자프로그람이 인용하지 않는 메쏘드이다. 체계의 Thread클라스에서 run( )메쏘드는 구체적인 내용을 가지지 않으므로 사용자프로그람은 Thread클라스의 하위클라스를 새로 창조해야 하며 run( )을 정의하여 원래의 run()메쏘드를 회복한다.

③ sleep( )메쏘드: 토막처리의 처리집행은 그 우선권에 따라 진행된다. 우선권이 높은 토막처리가 완성되지 않으면 즉 소멸되지 않으면 우선권이 낮은 토막처리는 처리기를 얻을 기회를 가지지 못한다. 어떤 경우에는 우선권이 낮은 토막처리가 일련의 작업을 한 다음 우선권이 높은 토막처리를 처리하여야 하는 경우가 있다. 또한 우선권이 높은 토막처리가 많은 시간을 들여 자기의 조작을 완성하여야 할 경우도 있다. 이때 우선권이 낮은 토막처리가 처리기를 차지하게 하자면 우선권이 높은 토막처리의 run( )메쏘드에서 sleep( )메쏘드를 호출하여 자체가 처리기자원을 포기하고 일정한 시간 휴식하게 해야 한다. 휴식시간의 길이는 sleep()메쏘드의 파라메터에 의해 결정된다.

sleep(int millsecond); //millsecond는 ㎳를 의미

sleep(int millsecond, int nanosecond); //nanosecond는 ㎱를 의미

④ isAlive( )메쏘드: stop( )메쏘드를 호출하여 토막처리를 중지하기전에 isAlive( )메쏘드로 이 토막처리가 아직 살아있는가를 한번 검사하여야 한다. 그것은 살아있는 토막처리가 체계오유를 발생시킬수 있기때문이다.

6.2.3. 프로그람에서 다중토막처리실현방법

프로그람에서 다중토막처리를 실현하자면 Thread클라스의 하위클라스를 창조하거나 Runable대면을 실현하여야 한다. 프로그람작성자가 어느 방법을 리용하든 반드시 사용자토막처리의 run( )메쏘드를 정의하고 적당한 시기 사용자토막처리실례를 창조하여야 한다.

1) Thread클라스의 하위클라스창조

사용자프로그람은 자체의 Thread클라스의 하위클라스를 창조하여야 하며 하위클라스에서 자기의 run( )메쏘드를 다시 정의하여야 한다. run( )메쏘드는 사용자토막처리의 조작을 포함하고있다. 이렇게 사용자프로그람이 자기의 토막처리를 만들어야 할 때 이미 정의한 Thread하위클라스의 실례를 창조하기만하면 된다.

실례 6-3

```
Example 6-3 TestThread.java
1: import java.io.* ;
2:
3: public class TestThread     //java Application주클라스
4: {
5:       public static void main(String args[])
6:       {
7:            if(args.length < 1)
8:            {
9:                 System.out.println("한개 지령행파라메터를 입력하십시오");
10:                System.out.exit(0);
11:           } //사용자정의의 thread하위클라스의 객체창조
12:           primeThread getPrimes = new primeThread(Integer.parseInt(args[0]));
13:           getPrimes.start( );
14:           while(getPrimes.isAlive( ) && getPrimes.ReadyToGoOn( ))
15:           {
16:                System.out.println("Counting the prime number...\n");
17:                  try
18:                  {
19:                       Thread.sleep(500);
20:                  }
21:                  catch(InterruptedException e)
22:                     { //sleep메쏘드가 일으킬수 있는 례외처리
23:                     return;
24:                }
```

```
25:            }
26:            System.out.println("임의의 건을 누르면 계속......");
27:            try{
28:                System.in.read( );
29:            }
30:            catch(IOException e) { }
31:        }
32:}
33:class primeThread extends Thread
34:{ //사용자정의의 Thread하위클라스창조
35:        boolean m_bContinue = true;
36:        int m_nCircleNum;
37:
38:        primeThread(int Num)
39:        {
40:            m_nCircleNum = Num;
41:        }
42:        boolean ReadyToGoOn( )
43:        {    return(m_bContinue); }
44:        public void run( )
45:        { //상위클라스thread를 계승하고 재정의하는 run메쏘드
46:            int number = 3;
47:            boolean flag = true;
48:
49:            while(true)
50:            {
51:                for(int i = 2; i < number; i++)
52:                if(number%i == 0)
53:                    flag = false;
54:                if(flag)
55:                    System.out.println(number + "는 씨수다");
56:                else
57:                    System.out.println(number + "는 씨수가 아니다");
58:                number++;
59:                if( number > m_nCircleNum )
60:                {
61:                    m_bContinue = false;
62:                    return;
63:                }
```

```
64:             flag = true;
65:             try
66:             {   //한번의 검사후 잠시 휴식
67:                 sleep(500);
68:             }
69:             catch(InterruptedException e)
70:             {
71:                 return;
72:             }
73:         }
74:     }
75:}
```

프로그람설명

이 프로그람은 Java Application으로서 2개의 클라스만을 정의하고있다. 하나는 프로그람의 주클라스 TestThread이고 다른 하나는 사용자정의의 Thread클라스의 하위클라스인 primeThread이다. 주클라스의 main( )메쏘드는 우선 사용자가 입력한 지령행파라메터에 따라 primeThread클라스의 객체를 창조하며 start( )메쏘드를 호출하여 이 하위토막처리객체를 기동하고 준비완료상태에 들어간다. 주토막처리는 우선 정보를 출력하여 자기가 활동상태에 있다는것을 표시하고 다음에 sleep( )메쏘드를 호출하여 자기는 일정한 시간 휴식하게 하며 하위토막처리가 처리기를 얻도록 한다.(주토막처리가 창조하는 하위토막처리는 우선권에 있어서 서로 같기때문에 만일 주토막처리가 집행중이라면 하위토막처리는 주토막처리의 집행이 끝난 후에야 처리기를 얻을수 있다.)

실행상태에 들어간 하위토막처리는 수값이 씨수인가를 검사하고 현시한 다음 상위토막처리가 처리기를 얻을수 있도록 일정한 시간 휴식상태에 들어간다. 처리기를 얻은 상위토막처리는 정보를 현시하여 자기가 활동상태에 있다는것을 표시하고 다음에 다시 휴식상태에 들어간다. 이렇게 매번 하위토막처리시동을 새롭게 하여 증가한 수값이 씨수인가를 검사하고 인쇄하며 이 수가 정한 값과 같아질 때까지 진행한다. 이때 하위토막처리는 run( )메쏘드로부터 귀환되여 이 실행을 결속하며 다음에 주토막처리가 결속된다.

프로그람의 실행결과를 그림 6-4에서 보여주었다.

사용자가 정의하는 클라스는 반드시 Thread클라스를 계승하여야 한다.

사용자하위클라스가 또 다른 상위클라스를 가지려면(실례로 Applet상위클라스) Java의 1-계층계승의 원칙에 따라 우의 수법을 적용할수 없다. 이때에는 다음의 방법(Runable대면의 실현)을 생각해 볼수 있다.

```
D:\Javatextbook\Test>java TestThread 5
Counting the prime number…

3는 씨수다
Counting the prime number…

4는 씨수가 아니다
Counting the prime number…

5는 씨수다
임의의 건을 누르면 계속……


D:\Javatextbook\Test>
```

그림 6-4. 실례 6-3의 실행결과

2) Runable대면의 실현

여기서는 상위클라스를 가지고있는 사용자클라스가 Runable대면을 실현하는 메쏘드를 통하여 사용자토막처리의 조작을 정의할수 있다. Runable대면은 오직 1개의 run( )메쏘드를 가지며 이 대면을 실현하려면 반드시 run( )메쏘드의 구체적내용을 정의해야 한다. 사용자가 토막처리를 새로 창조하는 조작 역시 이 메쏘드에 의해 결정된다. run( )을 정의한 후에 새로운 토막처리를 만들 때에는 run( )메쏘드를 실현하고있는 클라스를 파라메터로 하는 체계클라스 Thread의 객체를 창조하기만 하면 된다. 그러면 사용자가 실현하는 run( )메쏘드를 계승할수 있다.

실례 6-4

```
Example 6-4 UseRunnable.java
1: import java.applet.Applet;
2: import java.awt.*;
3: public class TestRunnable extends Applet implements Runnable
4: {
5:      Label prompt1 = new Label("첫번째 하위토막처리");
6:      Label prompt2 = new Label("두번째 하위토막처리");
7:      TextField threadFirst = new TextField(14);
8:      TextField threadSecond = new TextField(14);
9:      Thread thread1, thread2;    //두개의 Thread객체
10:     int count1 = 0, count2 = 0;    //두개의 계수기
11:
12:     public void init( ) //초기화
```

```
13:    {
14:        add(prompt1);
15:        add(threadFirst);
16:        add(prompt2);
17:        add(threadSecond);
18:    }
19:    public void start( )
20:    {   //토막처리창조, 문자렬로 토막처리객체의 이름을 지정
21:        thread1 = new Thread(this, "FirstThread");
22:        thread2 = new Thread(this, "SecondThread");
23:        thread1.start( );     //토막처리객체시동, 준비완료상태진입
24:        thread2.start( );
25:    }
26:    public void run( )     //Runnable 대면의 run( )메쏘드 실현
27:    {
28:        String currentRunning;
29:        while(true)     //무한순환
30:        {
31:            try
32:            {    //활성중인 토막처리가 0~3s 휴식하기
33:                Thread.sleep((int)(Math.random( )*30000));
34:            }
35:            catch(InterruptedException e){ }
36:            currentRunning = Thread.currentThread( ).getName( );
37:            if(currentRunning.equals("FirstThread"))
38:            {   count1++;
39:                threadFirst.setText("토막처리 1은 " + count1 + "번 일정짜기됨");
40:            }
41:            else if(currentRunning.equals("SecondThread"))
42:            {
43:                count2++;
44:                threadSecond.setText("토막처리 2는 " + count2 + "번 일정짜기됨");
45:            }
46:        }
47:    }
48:}
```

프로그람설명

실례 6-4의 프로그람은 Java Applet프로그람이다. 그러므로 프로그람의 주클라스 TestRunable은 반드시 Applet클라스의 하위클라스이여야 한다. 동시에 그것은 Runable대면을 실현하고 run( )메쏘드를 구체적으로 실현하고있다. TestRunable에서 2개의 하위토막처리를 창조하였는데 모두 Thread클라스의 객체이다. 이 두 객체의 구성자는 그것들이 현재클라스에서 계승하고있는 run( )메쏘드를 지정하고있다. 즉 그것들은 처리시 이 run( )메쏘드를 집행한다. run( )메쏘드는 우선 우연적인 시간으로 휴식하고 다음 활동하는 토막처리(즉 처리기를 얻은 토막처리)를 계수하여 회수를 상응한 본문칸에 현시한다.

그림 6-5는 실례 6-4의 실행결과이다.

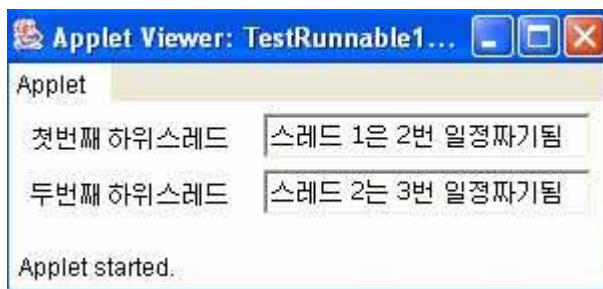



그림 6-5. 실례 6-4의 실행결과

제7장. Java의 입출력

외부설비나 다른 콤퓨터와의 교류를 진행하는 입출력조작은 하드디스크에 대한 파일조작으로써 프로그람에서 중요하고 반드시 있어야 하는 기능이다. 어떤 언어이든지 반드시 입출력에 대한 지원을 제공하고있다. Java언어 역시 례외가 되지 않으며 입출력클라스서고에는 이미 정의한 서로 다른 경우에 쓰이는 입출력클라스들이 포함되여있다. 그것들을 리용하여 Java프로그람은 여러가지 입출력조작과 복잡한 파일 및 등록부관리를 편리하게 실현할수 있다.

제1절. Java입출력클라스서고

- 기본입출력흐름클라스: InputStream, OutputStream
- 표준입출력의 객체: System.in, System.out

Java의 입출력기능은 반드시 입출력클라스서고 java.io패키지에 의하여 실현된다. 이 패키지에서 클라스의 대부분은 입출력의 흐름클라스를 완성하는데 리용된다. 아래에서 흐름의 개념을 먼저 소개한다.

7.1.1. 흐름의 개념

흐름은 콤퓨터의 입력과 출력사이에서 움직이는 자료의 순서렬을 말한다. 즉 입력흐름은 외부장치로부터 콤퓨터에 들어오는 자료순서렬을 의미하며 출력흐름은 콤퓨터로부터 외부장치에로 나가는 자료순서렬을 의미한다.

흐름입출력은 자주 보게 되는 입출력방식으로서 가장 큰 특징은 자료의 읽기와 쓰기가 자료순서렬에 따라 순차적으로 진행된다는것이다. 즉 매개 자료는 반드시 그 앞에 있는 자료가 읽어지거나 쓰기된 후에야 읽기쓰기될수 있다. 매번 읽기쓰기조작이 처리하는것은 순서렬에 남은 읽기쓰기하지 못한 자료의 첫째자료이며 마음대로 입출력위치를 선택할수 없다. 테프장치는 흐름식입출력을 실현하는 전형적인 장치이다.

흐름순서렬에서 자료는 아직 가공하지 못한 원시자료일수도 있고 이미 일정하게 코드처리한 후 어떤 형식에 맞게 규정한 특정한 자료일수도 있다.

7.1.2. 입출력흐름클라스

Java의 입출력클라스서고에서 매개 클라스는 구체적인 입력 또는 출력흐름을 나타내며 모든 입출력흐름클라스는 Java의 기본입출력흐름클라스에서 파생한 하위클라스이다.

가장 기본적인 흐름클라스에는 2개가 있다. 하나는 기본입력흐름 InputStream이고 다른 하나는 기본출력흐름 OutputStream이다. 다른 모든 입력흐름클라스는 InputStream을 계승한것으로서 자기의 특성에 맞게 기능들을 확장한 InputStream클라스의 하위클라스이다. 마찬가지로 다른 모든 출력흐름클라스 역시 OutputStream의 하위클라스이다.

1) InputStream클라스

InputStream은 입력흐름이 요구하는 모든 메쏘드들을 포함하며 입력흐름으로부터 자료를 읽어들이는 가장 기본적인 기능을 수행한다.

Java프로그람이 외부장치로부터 자료를 읽어들이자면 적당한 류형의 입력흐름클라스의 객체를 창조하여 외부장치 즉 건반, 하드디스크, 망소케트 등과의 접속을 실현하여야 한다. 다음으로 새로 창조된 흐름클라스객체를 실현하는 구체적인 메쏘드(례: read( ))를 호출하여 외부장치에 맞는 입력조작을 실현하여야 한다. 중요한것은 InputStream은 실례화할수 없는 추상클라스이므로 실제프로그람에서 창조한 입력흐름은 일반적으로 InputStream의 어떤 하위클라스의 객체이고 그에 의하여 외부장치와의 접속을 실현한다는것이다. 동시에 이 객체는 InputStream하위클라스의 실례로 되여 InputStream을 계승한 아래의 메쏘드를 사용할수 있다.

① 입력흐름으로부터 자료를 읽어들이는 메쏘드: read( )메쏘드는 InputStream이 입력흐름으로부터 자료를 읽어들이는 메쏘드이다. 3개의 서로 다른 read( )메쏘드가 있는데 공통적인 특징은 자료를 바이트단위로 읽어들인다는것이다. 즉 자료를 2진방식으로만 읽어들인다는것이다.

· public int read( ): 이 메쏘드는 입력흐름으로부터 한 바이트(8bit)의 2진자료를 읽어들이고 이 자료를 낮은 비트의 바이트로 하여 16bit의 옹근수형으로 되게 한 후 이 메쏘드를 호출한 명령문에 귀환시킨다. 만일 입력흐름의 현재위치에 자료가 없다면 -1을 귀환시킨다.

· public int read(byte b[]): 이 메쏘드는 입력흐름의 현재위치로부터 여러개의 바이트를 련속 읽어들여 파라메터가 지적한 바이트배렬 b[]에 보존한다. 동시에 읽어들인 바이트의 배렬을 귀환시킨다.

· public int read(byte b[], int off, int ren): 바이트배렬의 주어진 위치로부터 주어진 길이만큼 읽어들여 바이트배렬 b[]에 보존한다. 동시에 읽어들인 바이트의 배렬을 귀환시킨다.

② 입력위치지적자를 정하는 메쏘드: 입력흐름에서 중요한것은 읽어들이는 조작의 순차성을 정하는것이다. 그러므로 매개 흐름은 하나의 위치지적자를 가진다. 흐름조작에 따르는 집행자와 함께 위치지적자는 자동적으로 뒤로 이동하여 다음에 읽어들일 자료를 가리킨다. 위치지적자는 read( )메쏘드가 입력흐름으로부터 어느 자료를 읽겠는가를 결정한다.

InputStream에서 위치지적자를 조종하는데 쓰이는 메쏘드에는 몇가지 있다.

· public long skip(long n): 위치지적자가 현재의 위치로부터 뒤로 n개 바이트를

뛰여넘게 한다.

· public void mark( ): 현재의 위치지적자가 있는 곳에 표기를 한다.

· public void reset( ): 위치지적자를 표기된 위치에 귀환시킨다.

③ 흐름을 닫는 메쏘드: 입력흐름의 사용이 끝난 다음에는 아래의 메쏘드를 호출하여 그것을 닫는다.

public void close( ): Java프로그람과 외부장치와의 련결을 끊고 이 련결이 차지하고있던 체계자원을 해방한다.

2) OutputStream클라스

OutputStream은 출력흐름이 사용하여야 할 모든 메쏘드를 포함하고있다.

읽어들이는 조작과 마찬가지로 Java프로그람이 어떤 외부장치에 자료를 출력하여야 할 때는 새로운 출력흐름객체를 창조하여 이 외부장치와의 련결을 실현하여야 한다. 다음에 OutputStream이 제공하는 write( )메쏘드를 리용하여 순서대로 이 외부장치에 쓰기한다. OutputStream도 실례화할수 없는 추상클라스이므로 여기서 창조한 출력흐름객체는 어떤 OutputStream의 하위클라스에 속해야 한다.

① 자료를 쓰는 메쏘드: 입력흐름과 류사하게 출력흐름 역시 순차적인 쓰기조작이 기본특징으로 되고있다. 앞의 자료를 이미 외부장치에 쓰기하였을 때에야 뒤의 자료를 출력할수 있다. 출력흐름이 2진방식으로 련결되여있는 외부장치에 원시자료를 바이트단위로 쓰기할수 있다. 그리고 전달된 자료에 대한 서식이나 형변환을 완성할수 없다.

· public void write(int b): 파라메터 b의 낮은쪽 1byte를 출력흐름에 순차적으로 쓰기한다.

· public void write(byte b[]): 바이트배렬 b[]의 바이트전부를 출력흐름에 순차적으로 쓰기한다.

· public void flush( ): 완충흐름식으로 출력한다. 즉 write( )메쏘드가 쓰기한 자료를 출력흐름과 서로 련결되여있는 외부장치에 직접 전달하지 않고 우선 흐름의 완충구역에 잠시 보관하였다가 자료가 일정한 정도까지 루적되였을 때 외부장치에 대한 쓰기조작을 한번에 통일적으로 진행하여 자료전부를 외부장치에 쓰기한다.

이러한 처리는 콤퓨터의 외부장치에 대한 읽기쓰기회수를 낮추고 체계의 효률을 크게 높인다.

② 출력흐름을 닫는 메쏘드: 출구조작이 완성되였을 때 호출한다.

public void close( ): Java프로그람과 외부장치와의 련결을 끊고 이 련결이 차지하고있던 체계자원을 해방한다.

JDK 1.1이상부터는 InputStream과 OutputStream에서 2개의 추상기본클라스를 새로 추가하였다. Reader클라스와 Writer클라스인데 이 추상클라스들은 주로 문자자료흐름에 대한 입력과 출력조작을 진행한다. 실례로 Reader클라스에서 정의하는 read( )메쏘드는 입력흐름으로부터 하나의 char형문자를 취한다. 그리고 read(char[])메쏘드는 입력흐름으로부터 몇개의 문자를 읽어들여 문자배렬에 보존한다. Writer클

라스의 Writer(string)메쏘드는 한개의 문자렬의 모든 문자들을 문자출력흐름에 쓰기한다.

3) 몇개의 구체적인 입출력흐름

기본입출력흐름은 기본적인 입출력조작을 정의하고있는 추상클라스이며 Java프로그람에서 실제 사용하는것은 그것들의 하위클라스이다. 자료원천과 입출력사명에 따라 서로 다른 입출력흐름이 대응한다. 여기서 자주 리용하는것은 다음과 같다.

려파형입출력흐름인 FilterInputStream과 FilterOutputStream는 자료를 입출력하는 동시에 전송되는 자료에 대한 형이나 형식을 지정하는 변환을 진행할수 있다. 즉 2진바이트자료에 대한 리해와 코드작성변환을 실현할수 있다.

파일입출력흐름인 FileInputStream과 FileOutputStream은 주로 현재파일에 대한 순차적인 읽기쓰기조작을 진행한다.

관형입출력흐름인 PipedInputStream과 PipedOutputStream은 프로그람내부의 토막처리사이의 통신이나 다른 프로그람들사이의 통신을 실현한다.

바이트배렬흐름인 ByteArrayInputStream과 ByteArrayOutputStream은 기억기완충구역과의 동기읽기쓰기를 실현하며 CPU등록기의 읽기쓰기조작도 할수 있다.

순차입력흐름 SeguenceInputStream은 다른 입력흐름의 머리부와 끝을 서로 련결할수 있으며 완전한 입력흐름으로 만들수 있다.

그밖에 JDK 1.1에서 추상클라스 Reader와 Writer로부터 파생한 하위클라스들은 InputStream과 OutputStream에 대하여 바이트단위의 입출력을 문자단위의 입출력으로 변환시키므로 사용하기가 편리하다.

자료입출력흐름인 DataInputStream과 DataOutputStream은 각각 려파형입출력흐름 FilterInputStream과 FilterOutputStream의 하위클라스이다. 려파형입출력흐름의 가장 중요한 작용은 바로 자료원천과 프로그람사이에 려파처리단계를 추가한것이며 원시자료에 대하여 특정한 가공, 처리, 변환조작을 한다. 자료입출력흐름은 각각 DataInput와 DataOutput의 대면에서 정의하고 구체적인 기계에 무관계한 띠형식읽기쓰기조작을 실현하며 서로 다른 형자료에 대한 읽기쓰기를 실현하고있다.

DataInputStream흐름은 서로 다른 형자료에 대한 여러가지 읽기메쏘드를 정의하고있다.(례: readByte( ), readBoolean( ), readShort( ), readChar( ), readInt( ), readLong( ), readFloat( ), readDouble( ), readLine( ) 등) 마찬가지로 DataOutputStream도 서로 다른 자료형에 대한 쓰기메쏘드들을 정의하고있다.(례: writeByte( ), writeBoolean( ), writeShort( ), writeChar( ), writeInt( ), writeLong( ), writeFloat( ), writeDouble( ), writeChars( ) 등)

이밖에도 Java의 입출력클라스서고에 포함되는 흐름클라스들은 많다. 여기서는 간단한 소개만을 한다.

7.1.3. 표준입출력

Java프로그람이 외부자료원천과 자료교환을 하려면 먼저 입력과 출력클라스의 객체를 창조한 다음 이 자료원천과의 련결을 진행하여야 한다. 그런데 여기서도 례외상황을 가지고있다. 그것은 콤퓨터체계에 대한 문자대면부의 표준입출력장치에서 읽기쓰기조작을 진행할 때이다.

일반적인 체계에서 표준입력은 보통 건반이며 표준출력은 영상표시장치이다. Java프로그람은 문자대면부를 사용하여 체계표준입출력사이에 자료통신을 진행한다. 즉 건반으로부터 자료를 읽어들이거나 화면에 자료를 출력하는것은 흔히 볼수 있는 조작이며 Java체계에서는 사전에 2개의 흐름객체를 정의해놓고 각각 체계의 표준입력, 표준출력과의 련계를 맺는다. 그것들이 바로 System.in과 System.out이다.

System클라스의 모든 속성과 메쏘드는 모두 정적이다. 즉 호출시 클라스이름 System은 앞붙이로 리용된다. System.in과 System.out는 System클라스의 정적속성이며 각각 체계의 표준입력과 표준출력에 대응한다.

1) 표준입력

Java의 표준입력 System.in은 InputSystem클라스의 객체이다. 프로그람에서 건반으로부터 자료를 읽어들어야 할 때 System.in의 read( )메쏘드를 호출하기만 하면 된다. 실례로 아래의 명령문은 건반으로부터 1byte의 자료를 읽어들인다.

```
char ch = System.in.read( );
```

System.in.read( )명령문은 반드시 try블로크에 포함되여야 하며 try블로크뒤에는 IOException례외를 접수할수 있는 catch블로크가 있어야 한다.

```
try{
    ch = System.in.read( );
}
catch(IOException e)
{...}
```

System.in.read( )메쏘드를 리용하여 건반완충기로부터 1byte자료를 읽어들여 귀환시키는것은 16bit의 옹근수변수이다. 주의할것은 이 옹근수변수의 낮은 바이트가 실지 입력하는 자료이며 높은 바이트는 전부 령이다. 그밖에 InputStream클라스의 객체 System.in은 건반으로만 2진자료를 읽어들일수 있으며 이 비트정보를 옹근수, 날자, 류동소수점수, 문자렬 등 복잡한 자료형의 변수로 변환할수 없다.

건반완충기에 읽어들인 자료가 없을 때 System.in.read의 집행은 체계를 실행시켜 막기(block)상태에로 이전시킨다.

막기상태에서 현재흐름은 이 명령문위치에 머물러 있게 되며 프로그람은 더 실행할수 없다. 그러므로 사용자가 건반으로 자료를 입력할 때에야 실행을 계속해나갈수 있다. 프로그람에서는 어떤 때에 System.in.read( )명령문을 화면을 잠시 유지하는 목적으로도 리용한다. 실례로

```
//화면을 주시하여 관찰
System.out.println("Press any key to finish the program");
try{
     char test = (char)System.in.read( );
}
catch(IOException e)
{...}
```

2) 표준출력

Java의 표준출력 System.out는 출력흐름 PrintStream클라스의 객체를 인쇄하는 것이다. PrintStream은 려파형출력흐름클라스 FilterOutputStream의 하위클라스이며 여기서 화면에 서로 다른 형자료를 출력하는 메쏘드 print( )와 println( )을 정의하고 있다.

(1) println( )메쏘드

println( )메쏘드는 여러가지 재정의형식을 가진다.

public void println(형 변수나 객체);

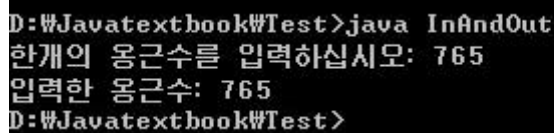
println( )의 작용은 어떤 파라메터가 지정하는 변수나 객체를 출력한 다음 다시 행을 바꾸어 유표가 화면의 1렬 첫번째 문자의 위치에 머물러있게 하는것이다. 만일 println( )메쏘드의 파라메터가 비여있으면 빈 행을 출력한다.

println( )메쏘드는 서로 다른 형의 변수나 객체들을 출력할수 있으며 boolean, double, float, int, long형의 변수 및 Object클라스의 객체를 포함하고있다. Java에서는 하위클라스객체가 실제파라메터로 되기때문에 상위클라스객체의 형식파라메터와 정합할수 있다. 또한 Object클라스는 모든 클라스의 상위클라스이므로 println( )은 실제적으로 재정의를 하여 클라스객체의 모든 화면출력을 실행할수 있다.

(2) print( )메쏘드

print( )메쏘드의 재정의는 println( )메쏘드와 완전히 같으며 화면에서 서로 다른 형의 변수와 객체를 출력하는 조작을 실현할수 있다. 다른점은 print( )메쏘드는 객체를 출력한 후 행바꾸기를 진행하지 않고 다음번 출력시 같은 행에 계속 출력한다는것이다.

실례 7-1

```
Example 7-1 InAndOut.java
1: import java.io.*;
2:
3: public class InAndOut
4: {
```

```
5:    public static void main(String args[])
6:    {
7:          try
8:          {
9:                BufferedReader br = new BufferedReader(
                                    new InputStreamReader(System.in));
10:         System.out.print("한개의 옹근수를 입력하십시오: ");
11:               int i = Integer.parseInt(br.readLine( ));
12:               System.out.print("입력한 옹근수: " + i);
13:         }
14:         catch(IOException e)
15:         {
16:               System.err.println(e.toString( ));
17:         }
18:   }
19:}
```

프로그람설명

실례 7-1의 프로그람은 9행에서 체계의 표준입력 System.in을 BufferReader객체로 변환한다. 11행은 BufferReader객체의 readLine( )메쏘드를 호출하여 표준입력으로부터 한행 문자를 읽어들이고 옹근수로 전환한다. 12행은 이 옹근수를 출력한다. 이 조작들이 일으킬수 있는 IOException례외는 catch블로크에 의해 보충되며 체계의 표준오유출력에 현시된다.

그림 7-1은 실례 7-1의 실행결과이다.

```
D:₩Javatextbook₩Test>java InAndOut
한개의 옹근수를 입력하십시오: 765
입력한 옹근수: 765
D:₩Javatextbook₩Test>
```

그림 7-1. 실례 7-1의 실행결과

제2절. 파일의 처리와 임의접근

- 등록부 및 파일관리클라스: File
- 파일입출력흐름클라스: FileInputStream, FileOutputStream
- 파일에 대한 임의접근클라스: RandomAccessFile

7.2.1. Java프로그람의 파일과 등록부관리

등록부는 파일을 관리하는 특수기구로서 같은 형의 파일을 같은 등록부에 보존하여 파일관리를 간단히 하고 작업효률을 높인다. Java언어는 파일관리를 지원할뿐아니라 등록부관리도 지원한다. Java언어에서는 등록부관리를 File클라스에 의하여 실현한다. File클라스도 java.io패키지에 있다. 그러나 InputStream이나 OutputStream의 하위클라스는 아니다. 때문에 그것은 자료의 입출력을 담당하지 않고 파일과 등록부를 관리하는데 전문적으로 쓰인다.

매 File클라스의 객체는 하나의 파일이나 등록부를 표시하며 이 파일이나 등록부에 대한 정보(례: 이름, 크기, 포함된 파일개수 등)가 객체의 속성으로 되고 그의 메쏘드를 호출하여 파일이나 등록부에 대한 조작을 진행할수 있다.(례: 창조, 삭제 등)

1) File클라스의 객체창조

매 File클라스의 객체는 파일이나 등록부에 대응된다. 그러므로 File클라스객체를 창조할 때 그에 대응하는 파일이나 등록부이름을 지정하여야 한다. File클라스는 모두 3개의 서로 다른 구성자를 제공하며 서로 다른 파라메터형식으로 파일과 등록부이름 정보를 접수한다.

(1) File(String path)

여기서 문자렬파라메터 path는 창조하려는 File객체에 대응하는 파일이나 등록부 및 그의 경로를 지정한다. 경로는 절대경로(례: "c:\myProgram\Sample.java"는 C구동기의 myProgram등록부의 파일 Sample.java를 표시한다.)일수도 있고 상대경로(례: "\myProgram\Sample . java"는 현재등록부의 보조등록부 myProgram의 파일 Sample.java를 표시한다.)일수도 있다. 만일 현재등록부가 C구동기의 뿌리등록부이면 우의 두 경로는 같은것으로 된다. 일반적으로는 상대경로를 사용한다.

path파라메터는 또한 원판의 어떤 등록부에 대응할수도 있다. (례: "c:\myProgram\Java"또는 "myProgram\Java") 여기서 주의하여야 할것은 조작체계에 따라 등록부분리부가 같지 않다는것이다. DOS나 Windows체계는 역사선을 사용하며 Unix체계는 바른사선을 사용한다. Java프로그람은 서로 다른 기반사이의 원활한 이식을 위하여 System클라스의 정적속성 System.dirSep를 사용한다. 이 속성에는 현재 체계가 규정하는 등록부분리부가 보존되여있으며 그것을 리용하여 경로를 합성할

수 있다. 실례로 우의 path는 "c:" + System.dirSep + "myProgram" + System.dirSep + "Java"로 쓸수 있다. 아래는 File객체를 창조하는 례제이다.

```
File f1 = new File("c:" + system.dirSep + "myProgram" + System.dirSep + "Java");
String s = "myProgram\Java";
File f2 = new File(s);
```

(2) File(String path, String name)

이 구성자는 2개의 파라메터를 가지는데 첫번째 파라메터 path는 파일이나 등록부에 대응하는 절대 또는 상대경로를 표시하며 두번째 파라메터 name은 파일이나 등록부이름을 표시한다. 경로와 이름을 구분하면 서로 같은 경로의 파일이나 등록부가 같은 경로문자렬을 공유할수 있기때문에 관리 및 수정이 비교적 편리하게 된다.

```
File f3 = new File("myProgram\Java", "FileIO.data");
```

(3) File(File dir, String name)

이 구성자의 첫번째 파라메터는 이미 존재하는 또 다른 File객체를 사용하여 파일이나 등록부의 경로를 표시하며 두번째 문자렬파라메터는 파일이나 등록부이름을 나타낸다.

```
String sdir = "myProgram" + System.dirSep + "Java";
String sfile = "FileIO.data";
File Fdir = new File(sdir);
File Ffile = new File(Fdir, sfile);
```

2) 파일이나 등록부의 속성얻기

어떤 원천파일이나 등록부에 대응하는 File객체를 창조하기만 하면 메쏘드를 호출하여 파일이나 등록부의 속성을 얻을수 있다.

여기서 자주 쓰이는 메쏘드는 다음과 같다.

· 파일이나 등록부가 존재하는가를 판단

public boolean exists( ); 파일이나 등록부가 존재하면 true, 아니면 false를 귀환시킨다.

· 파일인가 등록부인가를 판단

public boolean isFile( ); 객체가 유효파일이면 true를 귀환시킨다.

public boolean isDirectory( );객체가 유효등록부이면 true를 귀환시킨다.

·파일이나 등록부의 이름과 경로를 얻기

public String getName( ); 파일이름이나 등록부이름을 귀환시킨다.

public String getPath( ); 파일이나 등록부의 경로를 귀환시킨다.

· 파일의 크기얻기

public long length( ); 파일의 바이트수를 귀환시킨다.

· 파일의 읽기쓰기속성을 얻기

public boolean canRead( ); 파일이 읽기가능파일이면 true, 그렇지 않으면 false를 귀환시킨다.

public boolean canWrite( ); 파일이 쓰기가능파일이면 true, 그렇지 않으면 false를 귀환시킨다.

· 등록부의 파일들을 렬거

public String []list( ); 등록부의 모든 파일이름들을 문자렬배렬에 보존하고 귀환한다.

· 2개의 파일이나 등록부의 비교

public boolean equals(File f); 2개의 File객체가 서로 같으면 true를 귀환한다.

· 파일이나 등록부의 조작

파일의 이름바꾸기

public boolean renameTo(File newFile); 파일이름을 newFile에 대응하는 파일이름으로 바꾼다.

파일의 삭제

public void delete( ); 현재의 파일을 삭제한다.

등록부의 창조

public boolean mkdir( ); 현재등록부의 보조등록부를 창조한다.

실례 7-2에서 File클라스의 메쏘드들을 어떻게 사용하는가를 보여준다.

실례 7-2

```
Example 7-2 FileOperation.java
1: import java.io.*;
2: public class FileOperation
3: {
4:      public static void main(String args[])
5:      {
6:          try{  //표준입력을 InputStreamReader을 통하여 2진자료흐름
                                    으로부터 문자자료흐름으로 바꾸기
7:              BufferedReader in = new BufferedReader
8:                              (new InputStreamReader(System.in));
9:              String sdir = "c:\\temp";
10:             String sfile;
11:             File Fdir1 = new File(sdir);
12:             if(Fdir1.exists( ) && Fdir1.isDirectory( ))
13:             {
14:                 System.out.println("There is a directory" + sdir + "exists.");
15:                 for(int i = 0; i < Fdir1.list( ).length; i++)
16:                     System.out.println((Fdir1.list( ))[i]);
```

```
17:                 File Fdir2 = new File("c:\\temp\\temp");
18:                 if(!Fdir2.exists( ))
19:                     Fdir2.mkdir( );
20:                 System.out.println( );
21:                 System.out.println("Now the new list after create a new dir:");
22:                 for(int i = 0; i < Fdir1.list( ).length; i++)
23:                     System.out.println((Fdir1.list( ))[i]);
24:             }
25:             System.out.println("Enter a file name in this directory:");
26:             sfile = in.readLine( );
27:             File Ffile = new File(Fdir1, sfile);
28:             if(Ffile.isFile( ))
29:             {
30:                 System.out.println("File" + Ffile.getName( )
31:                         + "in Path" + Ffile.getPath( )
32:                         + "is"+Ffile.length( ) + "in length.");
33:             }
34:         }
35:         catch(Exception e) //입출력조작이기만 하면 례외를 발생
36:         {
37:             System.out.println(e.toString( ));
38:         }
39:     }
40:}
```

그림 7-2는 실례 7-2의 실행결과이다.

```
D:\Javatextbook\Test>java FileOperation
There is a directoryc:\tempexists.
temp
figure 8-16-b.bmp

Now the new list after create a new dir:
temp
figure 8-16-b.bmp
Enter a file name in this directory:
figure 8-16-b.bmp
Filefigure 8-16-b.bmpin Pathc:\temp\figure 8-16-b.bmpis249570in length.

D:\Javatextbook\Test>
```

그림 7-2. 실례 7-2의 실행결과

프로그람설명

실례 7-2의 7행은 체계표준입력으로부터 문자방식으로 자료를 읽어들이는 입력흐름객체 in을 창조한다. 11행은 File클라스의 객체를 창조하여 C구동기의 temp등록부를 가리킨다. 12-16행은 이 등록부가 존재하는 조건에서 여기에 포함되는 파일과 보조등록부를 렬거한다. 19행은 이 등록부에 temp보조등록부를 창조하며 22-23행에서 C구동기의 temp등록부안의 모든 파일과 보조등록부를 현시한다. 26행부터는 표준입력으로부터 파일이름을 읽어들이고 이 파일의 련관정보를 출력한다.

7.2.2. 파일입출력흐름

만일 파일로부터 자료를 읽어들이거나 자료를 파일에 쓰려면 파일입출력흐름클라스 FileInputStream과 FileOutputStream을 사용하여야 한다.

실례 7-3은 FileInputStream과 FileOutputStream을 리용하여 원판의 파일읽기쓰기를 완성하는 실례이다. 이 프로그람은 1개 파일을 창조하고 여기에 부분자료를 쓴 다음 이 자료들을 다시 읽어들여 검사한다.

실례 7-3

```
Example 7-3 MyFileIo.java
1: import java.io.*;
2: public class MyFileIo
3: {
4:      public static void main(String args[])
5:      {
6:          char ch;
7:          int chi;
8:          File MyPath = new File("\\temp");
9:          if(!MyPath.exists( ))
10:             MyPath.mkdir( );
11:         File MyFile1 = new File(MyPath, "crt.txt");
12:         try{
13:             FileOutputStream fout = new FileOutputStream(MyFile1);
14:             System.out.println("Input a String finished with # please:");
15:             while((ch = (char)System.in.read( )) != '#')
16:                 fout.write(ch);
17:             fout.close( );
18:             System.out.println("");
19:             FileInputStream fin = new FileInputStream(MyFile1);
20:             while((chi = fin.read( )) != -1)
21:                 System.out.print((char)chi);
```

```
22:            fin.close( );
23:        }
24:        catch(FileNotFoundException e){
25:            System.err.println(e);
26:        }
27:        catch(IOException e){
28:            System.err.println(e);
29:        }
30:    }
31:}
```

그림 7-3은 실례 7-3의 실행결과이다.

```
D:\Javatextbook\Test>java MyFileIo
Input a String finished with # please:
eyru#

eyru
D:\Javatextbook\Test>
UniK 半 :
```

그림 7-3. 실례 7-3의 실행결과

이 례제로부터 볼수 있는바와 같이 파일입출력흐름을 리용하여 파일을 완성하는 읽기쓰기는 일반적으로 아래의 단계를 거쳐야 한다.

① 파일이름문자렬과 File객체를 리용하여 입출력흐름객체를 창조한다.

FileInputStream은 두개의 구성자를 가지고있다.

· FileInputStream(String FileName)은 파일이름(경로이름을 포함)문자렬을 리용하여 이 파일로부터 자료를 읽어들이는 입력흐름을 창조한다.

· FileInputStream(File f)는 이미 존재하는 File객체를 리용하여 이 객체에 대응하는 파일로부터 자료를 읽어들이는 파일입력흐름을 창조한다.

주의하여야 할것은 어느 구성자이든 파일입력이나 출력흐름을 창조할 때 주어진 파일이름이나 경로가 틀리고 파일의 속성이 틀리면 파일을 읽어들일수 없고 오유가 생긴다는것이다. 이때 체계는 례외 FileNotFoundException을 통보할수 있다. 그러므로 파일입출력흐름을 창조하고 구성자를 호출하는 명령문은 try블로크에 포함되여야 하며 상응한 catch블로크를 가지고 그것들이 산생할수 있는 례외들을 처리하여야 한다.

② 파일입출력흐름으로부터 자료읽기쓰기

파일입출력흐름으로부터의 자료읽기쓰기에는 2가지 방식이 있다. 하나는 FileInputStream자체의 읽기쓰기기능을 직접 리용하는것이고 다른 하나는 FileInputStream과 FileOutputStream이 원시자료원천으로 되여 기타 기능이 비교적

간단한 입출력흐름에 다시 접속시켜 객체의 읽기쓰기조작을 완성하는것이다.

FileInputStream과 FileOutputStream자체의 읽기쓰기기능들은 상위클라스 InputStream과 OutputStream으로부터 직접 계승된것이다. 이것은 바이트단위로 된 원시 2진자료의 읽기쓰기만을 완성할수 있다. read( )와 write( )의 집행은 입출력오유에 의하여 IOException례외를 발생시키며 파일조작에서 read( )조작을 집행할 때 막기를 조성시킬수 있다.

서로 다른 형의 자료를 파일로부터 편리하게 읽기쓰기하기 위하여 주로 두번째 방식을 리용한다. 자료원천인 FileInputStream과 FileOutputStream을 자기파일과의 넘기기관계로 설정한 다음 다른 흐름클라스객체를 창조하여 FileInputStream과 FileOutputStream객체로부터 자료를 읽기쓰기한다. 일반적으로 비교적 자주 쓰이는것이 려파형흐름인 2개의 하위클라스 DataInputStream과 DataOutputStream으로서 아래의 메쏘드로 더 간단히 실현할수 있다.

```
File MyFile = new File("MyTextFile");
DataInputStream din = new DataInputStream(new FileInputStream(MyFile));
DataOutputStream dout = new DataOutStream(new FileOutputStream(MyFile));
```

7.2.3. 프로그람에서 파일에 대한 임의접근

FileInputStream과 FileOutputStream이 실현하는것은 파일에 대한 순차적인 읽기쓰기이며 읽기쓰기는 각각 서로 다른 객체를 창조하여야 한다. Java에서는 기능이 보다 풍부한 클라스 RandomAccessFile을 정의하여 파일에 대한 임의읽기쓰기조작을 실현하고있다.

RandomAccessFile객체의 창조;

- RandomAccessFile(String name, String mode);
- RandomAccessFile(File f, String mode);

우에서 보여준것은 RandomAccessFile클라스의 2개의 구성자이다. 어느것을 사용하여 RandomAccessFile객체를 창조하든지간에 2개의 파라메터를 제공할것을 요구한다. 첫번째 파라메터는 자료원천의 파일로서 파일이름문자렬이나 파일객체의 메쏘드로 표현되며 두번째 파라메터는 RandomAccessFile객체가 어떤 방식으로 지정한 파일에 접근할수 있는가를 결정한다. 접근방식에는 2가지가 있다. r는 읽기전용방식으로 파일을 열기한다. rw는 읽기쓰기방식으로 파일을 열기한다는것을 의미하는데 이때 하나의 객체는 동시에 읽기쓰기 두조작을 실현할수 있다.

RandomAccessFile객체를 창조할 때 2가지 례외가 생성될수 있다. 즉 지정된 파일이 존재하지 않을 때 체계는 FileNotFoundException을 던지며 읽기쓰기방식을 리용하여 읽기전용속성의 파일을 열려고 하거나 다른 입출력오유가 나타나면 IOException례외를 던질수 있다. 아래는 RandomAccessFile객체창조의 실례이다.

```
File BankMegFile = new File("BankFile.txt");
RandomAccessFile MyRAF = new RandomAccessFile(BankMegFile, "rw");
```

1) 파일위치지적자에 대한 조작

문자읽기쓰기조작과 달리 RandomAccessFile이 실현하는것은 임의읽기쓰기이다. 객체의 임의의 위치에서 자료읽기쓰기를 할수 있는데 꼭 앞에서 뒤로 진행해야 하는것은 아니다. 임의의 위치에서 이러한 기능을 실현하자면 반드시 파일위치지적자와 이 지적자를 이동시키는 메쏘드를 정의하여야 한다. RandomAccessFile객체의 파일위치지적자는 이러한 규칙에 따르고있다.

RandomAccessFile객체를 새로 창조한 후 파일위치지적자는 파일의 시작부에 위치한다.

매 읽기쓰기조작후 파일위치지적자는 읽기쓰기의 바이트수만큼 이동된다.

· getPointer( )메쏘드를 리용하면 파일머리부부터 현재파일위치지적자의 절대위치를 얻을수 있다.

```
public long getPointer( );
```

· seek( )메쏘드를 리용하면 파일위치지적자를 이동시킬수 있다.

```
public void seek(long pos);
```

이 메쏘드는 파일위치지적자를 파일머리부로부터 파라메터 pos가 지정한 절대위치에로 이동시킨다.

· length( )메쏘드는 파일의 바이트길이를 귀환시킨다.

length( )메쏘드가 귀환시키는 파일길이와 위치지적자를 비교하면 파일을 마지막까지 읽어들였는가를 판단할수 있다.

2) 읽기조작

DataInputStream과 류사하게 RandomAccessFile클라스 역시 DataInput대면을 실현하고있다. 그것 역시 여러 메쏘드를 리용하여 서로 다른 형의 자료를 각각 읽어들일수 있으며 FileInputStream에 비해 더 강한 기능을 갖추고있다. RandomAccessFile에서의 읽기메쏘드에는 readBoolean( ), readChar( ), readLong( ), readFloat( ), readDouble( ), readLine( ), readUTF( ) 등이 있다. readLine( )은 현재위치로부터 시작하여 처음으로 '\n'이 될 때까지 한행본문을 읽어들이며 String객체를 귀환시킨다. 다른 메쏘드들은 앞에서 이미 소개되였으므로 다시 서술하지 않는다.

3) 쓰기조작

RandomAccessFile클라스는 또한 DataOutput대면을 실현할수 있다. RandomAccessFile클라스가 포함하는 쓰기메쏘드에는 writeBoolean( ), writeChar( ), writeUTF( ), writeInt( ), writeLong( ), writeFloat( ), writeDouble( ) 등이 있다. 여기서 writeUTF( )메쏘드는 파일에 한개의 문자렬객체를 출력할수 있다.

주의하여야 할것은 RandomAccessFile클라스의 모든 메쏘드는 IOException례외를 던질수 있으므로 련관된 명령문들을 try블로크에 놓아야 하며 catch블로크에서 대응한 례외객체를 처리하여야 한다는것이다.

실례 7-4

```
Example 7-4 TestFileDialog.java
1: import java.io.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestFileDialog
6: {
7:     public static void main(String args[])
8:     {
9:         new FileFrame( );
10:    }
11:}
12:class FileFrame extends Frame implements ActionListener
13:{
14:    TextArea ta;
15:    Button open, quit;
16:    FileDialog fd;
17:
18:    FileFrame( )
19:    {
20:        super("본문파일을 얻고 현시");
21:        ta = new TextArea(10, 45);
22:        open = new Button("열기");
23:        quit = new Button("닫기");
24:        open.addActionListener(this);
25:        open.addActionListener(this);
26:        setLayout(new FlowLayout( ));
27:        add(ta);
28:        add(open);
29:        add(quit);
30:        setSize(350, 280);
31:        show( );
32:    }
33:    public void actionPerformed(ActionEvent e)
34:    {
35:        if(e.getActionCommand( ) == "열기")
36:        {
```

```
37:             fd = new FileDialog(this, "파일열기", FileDialog.LOAD);
38:             fd.setDirectory("c:\\temp"); //파일대화칸의  뿌리등록부 설정
39:             fd.show( );
40:             try{
41:                 File myfile = new File(fd.getDirectory( ), fd.getFile( ));
42:                 RandomAccessFile raf = new RandomAccessFile(myfile, "r");
43:                 while(raf.getFilePointer( ) < raf.length( ))
44:                 {
45:                     ta.append(raf.readLine( ) + "\n");
46:                 }
47:             }
48:             catch(IOException ioe)
49:             {
50:                 System.err.println(ioe.toString( ));
51:             }
52:         }
53:         if(e.getActionCommand( ) == "닫기")
54:         {
55:             dispose( );
56:             System.exit(0);
57:         }
58:     }
59:}
```

프로그람설명

실례 7-4는 도형사용자대면부의 Java Application이다. 이 프로그람은 객체를 리용하여 본문파일로부터 읽어들인 내용을 도형사용자대면부의 본문구역에 현시한다. 또한 파일대화칸 FileDialog를 사용하여 사용자가 매 등록부를 편리하게 탐색하며 하나의 파일을 선택할수 있게 한다.(그림 7-4) 파일대화칸에서 파일을 선택하여 열기전에는 프로그람의 다른 부분을 조작할수 없다.

일단 사용자가 선택하여 파일대화칸의 《열기》단추를 찰칵하면 파일대화칸은 자동적으로 닫기며 getFile( )메쏘드와 getDirectory( )메쏘드에 의해 선택된 파일의 이름과 등록부가 얻어진다. 파일대화칸의 창조는 3개의 파라메터를 요구한다. 첫번째 파라메터는 대화칸이 속한 Frame객체이고 두번째 파라메터는 창문표제이다. 마지막 파라메터는 두개의 상수를 가질수 있는데 FileDialong.LOAD는 파일을 열기 위한 목적으로 대화칸을 열고 FileDialong.SAVE는 파일을 보존할 목적으로 대화칸의 연다. 마지막으로 대화칸을 현시하려면 show( )메쏘드를 호출하여야 한다. 실례 7-4의 실행결과는 그림 7-4에서 보여준다.

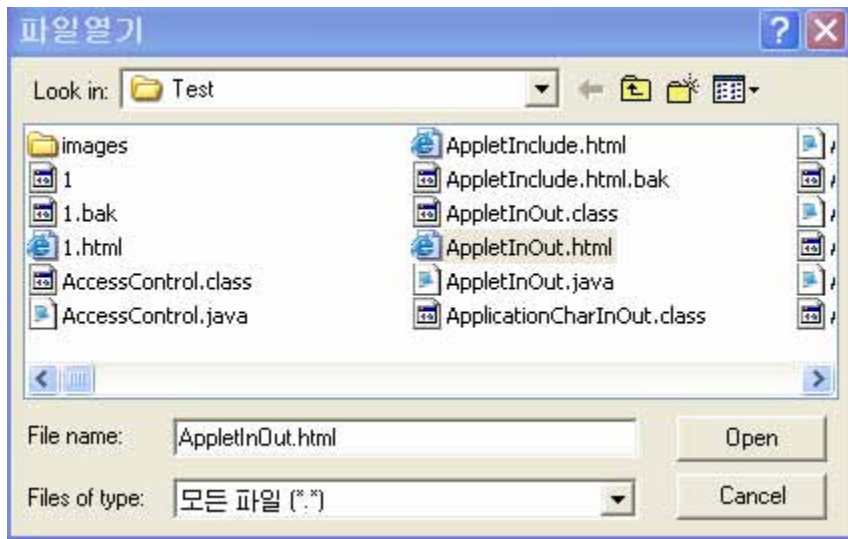



그림 7-4. 실례 7-4의 실행결과

제3절. 객체의 직렬화

- 객체의 직렬화는 객체를 다른 곳으로 이동시키는 Java기술이다.
- 객체직렬화의 실현방법
- class 클라스이름 implements Serializable
- 사용자정의의 직렬화코드전체의 실현
- class 클라스이름 implements Externalizable

이 절에서는 Java언어가 지원하는 기능중에서 객체를 저장가능한 속성으로 만들어 프로그람들사이에 객체를 주고받을수 있는 직렬화의 기능을 소개한다.

7.3.1. 객체의 직렬화란 무엇인가.

객체의 직렬화란 한마디로 객체를 다른 곳으로 이동시키는 Java기술을 말한다. 직렬화를 리용하면 객체의 내용을 파일에 보관할수도 있고 망을 통하여 다른 곳에 전송할수도 있으며 RMI에서의 객체통신도 가능하게 한다.

일반적으로 전송하여야 할 객체들의 형은 같지 않다. 만일 이러한 객체들의 정보를 파일에 보관하려면 먼저 매 객체의 형을 보관하고 다음에 자료를 저장하여야 한다. 또한 파일로부터 객체의 형을 읽은 다음 그 형의 객체를 창조하고 그 안에 내용들을 채워넣는 과정을 거쳐야 한다. 이러한 작업은 객체의 직렬화로 실현할수 있다.

1) 객체직렬화의 구성방식

객체를 전송할 때는 송신측과 수신측이 있다. 송신측은 객체를 차례로 보내고 즉 직렬화하며 수신측은 이것을 다시 재구성하여야 한다. 이것은 ObjectOutputStream과 ObjectInputStream으로 실현한다.

실례 7-5-1

```
Example 7-5-1 MySerial.java
1: import java.io.*;
2: class MySerial implements Serializable
3: {   public String name = "김영철";
4:      public int age = 20;
5:      public void myPrint( )
6:      {     System.out.println("NAME: " + name);
7:            System.out.println("AGE: " + age);
8:      }
9: }
```

실례 7-5-2

```
Example 7-5-2 MySerialWrite.java
1: import java.io.*;
2: import java.util.*;
3: public class MySerialWrite
4: {
5:      public static void main(String args[]) throws Exception
6:      { //객체를 파일로 보관
7:          FileOutputStream fos = new FileOutputStream("my.ser");
8:          ObjectOutputStream oos = new ObjectOutputStream(fos);
9:          oos.writeObject(new MySerial( ));
10:         oos.writeObject(new Date( ));
11:         oos.close( );
12:         fos.close( );
13:     }
14:}
```

실례 7-5-3

```
Example 7-5-3 MySerialRead.java
1: import java.io.*;
2: import java.util.*;
3: public class MySerialRead
4: {
5:      public static void main(String args[]) throws Exception
6:      {//파일내용을 읽어들여 객체로 생성
7:          FileInputStream fis = new FileInputStream("my.ser");
8:          ObjectInputStream ois = new ObjectInputStream(fis);
9:          MySerial my = (MySerial)ois.readObject( );
10:         Date d = (Date)ois.readObject( );
11:         ois.close( );
12:         fis.close( ); //객체직렬화를 통하여 객체의 성원을 출력
13:         my.myPrint( );
14:         System.out.println("Date: " + d);
15:     }
16:}
```

프로그람설명

이 실례는 MySerial이라는 클라스의 객체 MySerial과 API의 Date클라스의 객체 Date를 my . ser라는 파일에 저장하고 그 파일로부터 읽어들이는 실례이다 . MySerialWrite클라스는 ObjectOutputStream의 writeObject( )를 통하여 객체들을 my.ser파일에 저장한다. MySerialRead클라스는 ObjectInputStream의 readObject( )를 통하여 my.ser파일로부터 객체들을 읽어들인다. ReadObject( )의 귀환값은 객체형이므로 강제형변환(casting)하여야 한다.

```
My.myPrint( );
System.out.println("Date:" + d);
```

이 두 명령문을 리용하여 객체들이 그대로 전송되였다는것을 확인할수 있다.

그림 7-5는 실례 7-5의 실행결과이다.

```
D:\Javatextbook\Test>java MySerialWrite

D:\Javatextbook\Test>java MySerialRead
NAME: 김명철
AGE: 20
Date: Mon Mar 29 16:44:17 PST 2004
```

그림 7-5. 실례 7-5의 실행결과

2) 객체의 흐름

객체직렬화에서 객체의 전송은 흐름을 통하여 실현한다. 실례 7-5에서 매 객체들은 ObjectInputStream, ObjectOutputStream의 메쏘드들을 리용하여 읽기쓰기한다.

· writeObject( ): ObjectOutputStream의 메쏘드로서 객체를 흐름에 쓰기하는데 사용된다. 그러나 객체가 implements Serializable로 되였는가를 먼저 확인하여야 한다.

· readObject( ): ObjectInputStream의 메쏘드로서 객체를 읽기전에 입력된 객체형의 클라스를 찾아 기정인 구성자를 리용하여 실례를 먼저 생성한다. 그 다음에 매개 값들을 읽어들인다.

흐름에는 또한 자료를 읽고쓰기 위한 메쏘드를 정의하는 DataInput, DataOutput대면이 있다. 그리고 이것들을 계승하고있는 ObjectInput, ObjectOutput대면들도 객체를 읽고쓰기 위한 메쏘드들을 정의한다. 실례 7-5에서 사용한 ObjectOutputStream, ObjectInputStream이 바로 ObjectInput, ObjectOutput대면을 실현한 클라스이다.

그러나 모든 객체들이 직렬화될수 있는것이 아니다. 실례 7-5의 MySerial클라스 선언부에서 알수 있는것처럼 Serializable이라는 대면을 구현한 클라스만이 가능한것이다. 만일 이 대면을 구현하지 않으면 객체직렬화를 할수 없다.

객체직렬화의 명령문형식은 아래와 같다.

class 클라스이름 implements Serializable

Serializable대면은 어떠한 메쏘드도 가지지 않으며 serialVersionUID변수만을 가지고 클라스가 직렬화될수 있는 객체인가를 표시해주는 역할만을 한다.

Serializable을 구현한 클라스의 하위클라스 역시 직렬화할수 있는 객체로 된다.

다음의 실례를 고찰하자.

실례 7-6-1

```
Example 7-6-1 SubMySerial.java
1: class SubMySerial extends MySerial
2: {
3:     public String address = "평양";
4:     public void submyPrint( )
5:     {
6:         myPrint( );
7:         System.out.println("ADDRESS:" + address);
8:     }
9: }
```

실례 7-6-2

```
Example 7-6-2 SubMySerialWrite.java
1: import java.io.*;
2: public class SubMySerialWrite
3: {
4:      public static void main(String args[]) throws Exception
5:      { //객체를 파일로 보관
6:          FileOutputStream fos = new FileOutputStream("submy.ser");
7:          ObjectOutputStream oos = new ObjectOutputStream(fos);
8:          oos.writeObject(new SubMySerial( ));
9:          oos.close( );
10:         fos.close( );
11:     }
12:}
```

실례 7-6-3

```
Example 7-6-3 SubMySerialRead.java
1: import java.io.*;
2: public class SubMySerialRead
3: {
4:      public static void main(String args[]) throws Exception
5:      { //파일의 내용을 읽어들여 객체로 생성
6:          FileInputStream fis = new FileInputStream("submy.ser");
7:          ObjectInputStream ois = new ObjectInputStream(fis);
8:          SubMySerial sp = (SubMySerial)ois.readObject( );
9:          ois.close( );
10:         fis.close( );
11:         sp.submyPrint( ); //객체직렬화를 통하여 객체의 성원을 출력
12:     }
13:}
```

프로그람설명

이 실례에서 SubMySerial클라스는 implements Serializable로 선언하지 않았어도 MySerial클라스를 계승하였기때문에 객체의 직렬화가 가능하게 된다. 실례로

Java AWT의 Frame객체를 객체입출력을 통해 원판에 보관했다가 다시 화면에 출력할수도 있다.

transient

클라스선언부가 implements Serializable로 되였다고 하여 그 객체의 모든 자료가 직렬화되는것은 아니다. 직렬화할수 없는 성원자료가 있거나 Stream에 쓰고싶지 않은 성원자료가 있다면 transient를 사용하여 제외할수 있다.

transient예약어를 성원변수앞에 선언하면 writeObject( )로 그 변수의 자료를 흐름에 쓰기하는것을 막을수 있다. 즉 transient로 선언된 자료는 객체직렬화에서 제외된다. 실례 7-5의 MySerial클라스의 속성 age를 아래와 같이 선언하면 즉

public transient int age;

로 하면 age의 값은 객체직렬화에서 제외된다. 객체가 복귀될 때 직렬화에서 제외된 자료들은 원래 값으로 남아있게 된다.

7.3.2. Applet와 직렬화

앞에서는 객체를 파일에 읽고쓰기하는것을 고찰하였다. 여기에서는 Applet객체를 파일에 쓰고 열람기에서 읽어보는 방법을 고찰한다. API에서 Applet클라스는 Component클라스의 하위클라스이므로 implements Serializable을 간접적으로 선언하고있는것으로 볼수 있으며 따라서 객체직렬화가 가능하게 된다.

HTML파일에서 사용하는 <APPLET>꼬리표에서 클라스파일대신 직렬화된 객체파일을 사용하기 위해 CODE라는 속성대신 Object속성을 사용한다. 확장자역시 .class가 아닌 .ser로 표현해야 한다. 다음의 실례를 고찰하자.

실례 7-7-1

```
Example 7-7-1 AppletSerialization.java
1: import java.applet.*;
2: import java.awt.*;
3: public class AppletSerialization extends Applet
4: {
5:     public void paint(Graphics g)
6:     {
7:         g.drawString("AppletSerializable Test", 100, 100);
8:     }
9: }
```

실례 7-7-2

```
Example 7-7-2 AppletWrite.java
1: import java.io.*;
2: public class AppletWrite
3: {
4:      public static void main(String args[]) throws Exception
5:      { //Applet객체를 파일로 보관
6:           FileOutputStream fos = new FileOutputStream("AppletSerialization.ser");
7:           ObjectOutputStream oos = new ObjectOutputStream(fos);
8:           oos.writeObject(new AppletSerialization( ));
9:           oos.close( );
10:          fos.close( );
11:     }
12:}
```

프로그람설명

AppletSerialization클라스는 g.drawString("Applet Serializable Test", 100, 100)을 통하여 화면에 《AppletSerializable Test》라는 문자렬을 현시한다.(그림 7-6) Appletwrite 클라스는 oos.writeObject(new AppletSerializable( ))을 통하여 Appletclass(AppletSerialization)객체를 AppletSerialization.ser라는 파일에 보관한다.

<APPLET object=AppletSerialization.ser width=300 height=300>

</APPLET>

이 명령문에 의해 객체를 직렬화시켜 보관해놓은 파일로부터 객체를 읽어들여 다시 원래의 상태로 회복하고있다.

그림 7-6. 실례 7-7의 실행결과

7.3.3. 소케트를 통한 객체의 직렬화

망을 통하여 객체를 전송하는 실례를 고찰하자. 소케트를 생성한 다음 ObjectInputStream과 ObjectOutputStream의 readObject( ), writeObject( )를 리용하여 망에서 객체를 읽고쓰기할수 있다. 이것 역시 직렬화할 때 반드시 객체를 Serializable로 구현하여야 한다.

아래의 실례는 봉사기에서 String객체를 망을 통하여 의뢰기에 보내고 이 String 객체를 받은 의뢰기는 그 문자렬을 화면에 출력한다.

실례 7-8-1

```
Example 7-8-1 Client.java
1: import java.io.*;
2: import java.net.*;
3: public class Client
4: {
5:      private int Port = 1234;
6:      private Socket s;
7:      private ObjectInputStream ois;
8:      private Object o;
9:      public Client( ) throws IOException, ClassNotFoundException
10:     {//봉사기에 접속을 위한 소케트를 생성
11:         s = new Socket("localhost", Port);
12:         //봉사기로부터 객체를 받아들일 흐름생성
13:         ois = new ObjectInputStream(s.getInputStream( ));
14:         //흐름을 통해 봉사기로부터 객체를 읽어들임
15:         o = ois.readObject( );  //받은 객체를 화면에 출력
16:         System.out.println("client start...");
17:         System.out.println(o);
18:     }
19:     public static void main(String args[])
20:     {
21:         try{
22:             new Client( );
23:         }
24:         catch(IOException e1)
25:         {
26:             System.err.println("IOException");
27:         }
```

```
28:         catch(ClassNotFoundException e2)
29:         {
30:             System.err.println("ClassNotFoundException");
31:         }
32:     }
33:}
```

실례 7-8-2

```
Example 7-8-2 Server.java
1: import java.net.*;
2: import java.io.*;
3: public class Server
4: {
5:      private int Port = 1234;
6:      private ServerSocket server;
7:      private Socket socket;
8:      private ObjectOutputStream oos;
9:      private String s = "hello";
10:     public Server( ) throws IOException
11:     {
12:         server = new ServerSocket(Port);
13:         System.out.println("Server Running ....port" + Port);
14:         while(true) //의뢰기의 요청을 무한순환상태로 대기
15:         {     socket = server.accept( );
16:               System.out.println("Connection from: " + socket.getInetAddress( ).
17:               getHostName( ));      //객체직렬화를 위한 출력흐름을 생성
18:               oos = new ObjectOutputStream(socket.getOutputStream( ));
19:               oos.writeObject(s); //객체의 직렬화
20:         }
21:     }
22:     public static void main(String args[])
23:     {
24:         try{
25:             new Server( );
26:         }
27:         catch(IOException e)
28:         {
29:             System.err.println("IOExeception Error");
```

```
30:         }
31:     }
32:}
```

프로그람설명

이 실례의 Server클라스에서는 oos.writeObject(s)를 통하여 문자렬《hello》를 객체직렬화하여 망으로 전송하였고 Client클라스에서는 o=ois.readObject( )를 통하여 String객체를 전송받아 System.out.println(o)을 사용하여 화면에 《hello》라는 문자렬을 출력한다.

실례 7-8의 결과는 그림 7-7에서 보여준다.

```
D:\Javatextbook\Test>java Server
Server Running ....port1234
Connection from: localhost

D:\Javatextbook\Test>java Client
client start...
hello
```

그림 7-7. 실례 7-8의 실행결과

7.3.4. 사용자정의의 객체직렬화

앞에서는 직렬화를 구현한 클라스의 하위클라스 역시 객체직렬화를 할수 있다는 것을 보았다. 여기에서는 implements Serializable로 되여있지 않은 클라스를 계승한 하위클라스의 객체에 대한 직렬화를 고찰한다. 상위클라스로부터 계승한 마당들은 개발자가 직접 직렬화하여야 한다. 사용자정의의 직렬화를 하려면 private메쏘드를 리용하여야 한다.

```
private void writeObject(Java.io.ObjectOutputStream out) throws IOException
private void readObject(Java.io.ObjectInputStream in) throws IOException,
                                                  ClassNotFoundException
```

readObject( )와 writeObject( )메쏘드들은 정확하게 선언하여야 한다. 그것은 객체의 직렬화시 번역기가 이 메쏘드들을 대조하여 직렬화코드를 집행하기때문이다.

사용자정의의 private메쏘드를 구현하면 ObjectOutputStream , ObjectInputStream클라스의 readObject( ), writeObject( )가 실행되는 대신에 사용자가 직접 구현한 readObject(ObjectInputStream in), writeObject(ObjectOutputStream out)가 집행된다. 또한 기본적인 직렬화작업을 진행하려면 우의 2개의 private메쏘드에서 반드시 defaultWriteObject( ), defaultReadObject( )를 호출하여야 한다.(실례 7-9)

실례 7-9

```
Example 7-9 SubMySerial1.java
1: import java.io.*;
2: class SubMySerial1 extends MySerial implements Serializable
3: {
4:      public String address;
5:      public SubMySerial1( ){ }
6:      public SubMySerial1(String name, int age, String address)
7:      {
8:          this.name = name;
9:          this.age = age;
10:         this.address = address;
11:     }
12:     private void writeObject(ObjectOutputStream out) throws IOException
13:     {
14:         out.defaultWriteObject( );
15:         out.writeObject(name);
16:         out.writeInt(age);
17:     }
18:     private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
19:     {
20:         in.defaultReadObject( );
21:         name = (String)in.readObject( );
22:         age = in.readInt( );
23:     }
24:     public void submyPrint( )
25:     {
26:         myPrint( );
27:         System.out.println("ADDRESS: " + address);
28:     }
29:}
```

프로그람설명

이 실례에서는 사용자정의의 private메쏘드를 리용하여 name, age의 값인 《김영철》과 20을 직렬화하여 읽고쓰는 코드를 작성하였다. 결과를 보면 null이나 0이 아닌 직렬화된 값이 나오는것을 알수 있다.(그림 7-8)

```
D:₩Javatextbook₩Test>java SubMySerial1Write

D:₩Javatextbook₩Test>java SubMySerial1Read
NAME: 김영철
AGE: 20
ADDRESS: null
```

그림 7-8. 실례 7-9의 실행결과

이번에는 우의 실례와 같이 부분적으로 사용자정의의 객체직렬화를 구현하는것이 아니라 직렬화코드전체를 사용자정의로 구현하는 방법을 보기로 하자. 이것은 Externalizable대면을 리용하여 실현할수 있다.

Externalizable의 선언부는 다음과 같다.

public interface Externalizable extends Java.io.Serializable

Externalizable을 사용할 때는 앞에서처럼 직렬화 할 객체의 선언부에서 implements Serializable대신 implements Externalizable로 해준다.

Externalizable대면을 구현하기 위해서는 아래의 2가지 메소드를 재정의하여야 한다.

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
public void writeExternal(ObjectOutput out) throws IOException

다음의 실례를 고찰하자.

실례 7-10-1

```
Example 7-10-1 MySerial1.java
1: import java.io.*;
2: class MySerial1 implements Externalizable
3: {
4:     public String name;
5:     public int age;
6:     public MySerial1( ){ }
7:     public MySerial1(String name, int age)
8:     {
9:         this.name = name;
10:        this.age = age;
11:    }
12:    public void writeExternal(ObjectOutput out) throws IOException
13:    {
14:        out.writeObject(name);
15:        out.writeInt(age);
```

```
16:    }
17:    public void readExternal(ObjectInput in) throws IOException,
                                                  ClassNotFoundException
18:    {
19:        name = (String)in.readObject( );
20:        age = in.readInt( );
21:    }
22:    public void myPrint( )
23:    {
24:        System.out.println("NAME: " + name);
25:        System.out.println("AGE: " + age);
26:    }
27:}
```

실례 7-10-2

```
Example 7-10-2 MySerial1Write.java
1: import java.io.*;
2: public class MySerial1Write
3: {
4:     public static void main(String args[]) throws Exception
5:     { //객체를 파일로 보관
6:         FileOutputStream fos = new FileOutputStream("my.ser");
7:         ObjectOutputStream oos = new ObjectOutputStream(fos);
8:         oos.writeObject(new MySerial1("김영철", 20));
9:         oos.close( );
10:        fos.close( );
11:    }
12:}
```

실례 7-10-3

```
Example 7-10-3 MySerial1Read.java
1: import java.io.*;
2: public class MySerial1Read
3: {
4:     public static void main(String args[]) throws Exception
```

```
5:    {//파일의 내용을 읽어들여 객체로 생성
6:        FileInputStream fis = new FileInputStream("my.ser");
7:        ObjectInputStream ois = new ObjectInputStream(fis);
8:        MySerial1 p = (MySerial1)ois.readObject( );
9:        ois.close( );
10:       fis.close( );
11:       p.myPrint( ); //객체의 직렬화를 통해 객체의 성원을 출력
12:   }
13:}
```

프로그람설명

이 실례는 writeExternal( ), readExternal( )메쏘드를 리용하여 모든 객체의 직렬화를 실현하고있다. 결과는 그림 7-9에서 보여준다.

```
D:₩Javatextbook₩Test>javac MySerial1*.java

D:₩Javatextbook₩Test>java MySerial1Write

D:₩Javatextbook₩Test>java MySerial1Read
NAME: 김영철
AGE: 20
```

그림 7-9. 실례 7-10의 실행결과

제8장. 도형사용자대면부의 설계와 실현

이 장에서는 도형사용자대면부를 설계하고 실현하는 방법을 소개한다. 도형사용자대면부는 프로그람과 사용자가 서로 대화하는 창문이다. Java프로그람에서는 자기의 도형사용자대면부를 설계작성하고 그것을 리용하여 사용자의 입력을 접수할수 있으며 또한 사용자에게 프로그람실행결과를 출력할수 있다. 이 장에서는 도형사용자대면부의 기본구성과 주요조작들인 도형그리기, 동화상현시, AWT패키지를 사용한 패키지의 부품과 Java를 실현하는 사건처리기능 등을 소개한다.

제1절. 도형사용자대면부

- 도형사용자대면부를 생성하는 패키지는 java.awt이다.
- 도형사용자대면부의 3대요소: 용기, 조종부품, 사용자정의성분

사용자대면부를 설계하고 구성하는것은 쏘프트웨어개발에서 하나의 중요한 작업으로 된다. **사용자대면부**는 사용자가 콤퓨터체계와 서로 대화하는 대면부이다. 도형사용자대면부(graphics user interface) 즉 GUI는 도형방식을 사용하여 사용자가 콤퓨터체계에 명령을 편리하게 보낼수 있게 해주고 기동조작과 체계실행의 결과를 도형방식으로 사용자에게 현시한다. 도형사용자대면부의 보급과 대면부요소의 표준화정도가 높아짐에 따라 도형사용자대면부를 실현하는 메쏘드와 도구 역시 상응하게 많이 출현하였다.

도형사용자대면부를 작성하는데 쓰이는 클라스서고는 java.awt패키지이다. awt는 abstract window toolkit(추상창문도구배렬)의 략자이다. awt클라스서고에서의 각종 조작은 가상적으로 존재하는 《추상창문》에서 진행되는것으로 본다.

도형사용자대면부의 설계와 실현은 다음과 같다.

· 대면부를 구성하는 매 성분과 요소를 창조하고 그것들의 속성과 위치관계를 지정하며 구체적인 요구에 따라 그것들을 배렬한다. 즉 도형사용자대면부의 물리적외형을 구성한다.

· 도형사용자대면부의 사건과 매 대면부요소의 서로 다른 사건에 대한 응답을 정의하여 도형사용자대면부와 사용자와의 호상교류를 실현한다.

Java에서 도형사용자대면부의 매개 요소와 성분을 구성하는데는 3가지가 있다. 즉 용기, 조종부품, 사용자정의성분이 있다.

1. 용기

용기는 다른 대면부성분과 요소를 조직하는데 쓰이는 단위이다. 일반적으로 말하여 응용프로그람의 도형사용자대면부는 우선 하나의 복잡한 용기(례: 창문)에 대응한다. 이 용기내부에는 많은 대면부성분과 요소를 포함하게 되며 이 대면부요소들자체가 역시 하나의 용기이다. 이 용기는 또 자기의 대면부성분과 요소들을 가지며 이러한 수법으로 복잡한 하나의 도형대면부체계를 구성한다.

2. 조종부품

용기와 달리 **조종부품**은 도형사용자대면부의 최소단위로서 그 안에는 더 다른 성분을 포함할수 없다. 조종부품은 사용자와의 직접적인 접촉을 실현하며 한개 명령(례: 차림표명령)을 가지고 사용자의 본문이나 선택입력을 접수하여 사용자에게 본문이나 도형을 현시해주는 작용을 한다. 말하자면 조종부품은 도형사용자대면부표준화의 결과이다. 현재 자주 쓰는 조종부품에는 단일선택단추, 다중선택단추, 내리떨구기목록, 본문칸, 본문구역, 지령단추, 차림표 등이 있다. 여기에서 본문칸, 단추와 표식자는 앞에서 사용한적이 있는 GUI부품들이다.

조종부품을 사용하자면 보통 다음의 단계를 거쳐야 한다.

· 조종부품클라스의 객체를 창조하고 그의 크기 등 속성을 지정한다.

· 어떤 배치방안을 리용하여 조종부품객체를 용기의 지정위치에 추가한다.

· 이 부품객체를 그것이 일으킬수 있는 사건에 대응하는 사건감시자에게 등록하고 사건처리메쏘드를 재정의함으로써 사용자와의 호상작용을 실현한다.

용기 역시 조종부품으로 볼수 있으므로 다른 용기의 내부에 포함될수 있다.

3. 사용자정의성분

표준도형대면부요소외에 사용자의 요구에 따라 사용자가 정의하는 도형대면부성분을 설계할수 있다.(례: 도형그리기, 표지도안사용 등) 사용자정의성분은 표준대면부요소처럼 체계에 의하여 식별되고 허용될수 없고 사용자의 동작에 응답할수 없으며 또한 호상작용기능을 갖추고있지 않다.

제2절. 사용자정의성분들

- 도형그리기클라스: Graphics
- 색갈조종클라스: Color
- 서체현시효과클라스: Font
- 화상현시메쏘드: drawImage( )

이 절에서는 서고클라스와 그의 메쏘드들을 리용하여 사용자가 정의한 도형대면부성분을 어떻게 그리는가를 소개한다.

8.2.1. 도형그리기

Graphics클라스를 리용하여 직선, 각종 4각형, 다각형, 원과 타원 등을 그릴수 있다. 아래의 례제는 이 메쏘드들을 종합적으로 보여주고있다.

실례 8-1

```
Example 8-1 DrawFigures.java
1: import java.awt.*;
2: import java.applet.Applet;
3: public class DrawFigures extends Applet
4: {
5:      public void paint (Graphics g)
6:      {
7:          g.drawLine(30, 5, 40, 5);              //선그리기
8:          g.drawRect(40, 10, 50, 20);            //직4각형그리기
9:          g.fillRect(60, 30, 70, 40);            //직4각형채우기
10:         g.drawRoundRect(110, 10, 130, 50, 30, 30);  //모가 없는 4각형그리기
11:         g.drawOval(150, 120, 70, 40);          //타원형그리기
12:         g.fillOval(190, 160, 70, 40);          //타원형채우기
13:         g.drawOval(90, 100, 50, 40);
14:         g.fillOval(130, 100, 50, 40);
15:         drawMyPolygon(g);                      //사용자정의의 다각형그리기
16:         g.drawString("They are figures!", 100, 220);
17:     }
18:     public void drawMyPolygon(Graphics g)
19:     {
20:         int[] xCoords = {30, 50, 65, 119, 127};
```

```
21:         int[] yCoords = {100, 140, 127, 169, 201};
22:         g.drawPolygon(xCoords, yCoords, 5);
23:     }
24:}
```

프로그람설명

그림 8-1은 실례 8-1의 실행결과이다.

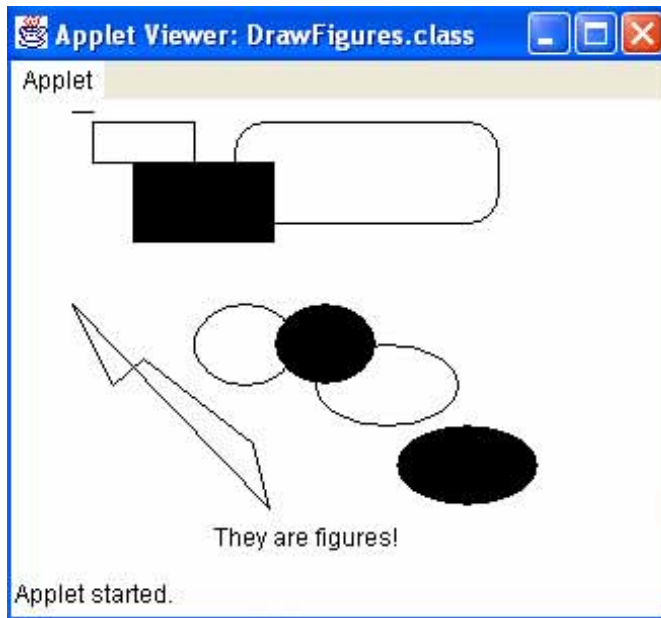



그림 8-1. 실례 8-1의 실행결과

실례에서 알수 있는것처럼 Java프로그람에서는 도형사용자대면부의 용기에서 도형을 그릴수 있다. 우선 그리려는 도형이 무엇인가 즉 원인가, 타원인가, 아니면 직선인가를 명확히 정해야 한다. 다음으로 도형이나 문자를 그리는 크기와 위치를 지정하여야 한다. 이것은 대면부용기에 대한 2차원화소자리표를 가지고 결정하여야 한다. Java에서 화면자리표는 화소를 단위로 하며 용기의 왼쪽우의 모서리는 가로자리표(x축)와 세로자리표(y축)의 원점으로 되고 오른쪽으로 가면서 그리고 아래로 내려가면서 자리표값이 증가한다.

도형을 그리는 메쏘드들은 자기의 능동적인 사용방식을 가진다. Graphics클라스외에 기하도형을 표시하는데 쓰이는 클라스들이 있다. 실례로 Point를 리용한 한개 화소점의 표시, Dimension클라스를 리용한 너비와 높이의 표시, Rectangle클라스를

리용한 한개 4각형의 표시, Polygon클라스를 리용한 다각형의 표시, Color클라스를 리용한 색갈의 표시 등이다.

8.2.2. 문자현시

Java에는 Font라는 클라스가 있으며 그것을 사용하면 보다 풍부하고 다채로운 서체 현시효과를 얻을수 있다.

Font클라스의 객체는 서체현시효과를 보여주는 객체로서 여기에는 서체이름, 서체의 격식 및 크기가 포함된다. 아래에 Font클라스의 객체를 창조하는 명령문을 보여준다.

```
Font MyFont = new Font("TimesRoman", Font.BOLD, 12);
```

MyFont에 대응하는것은 12pt, TimesRoman형의 강조체활자이다. 여기서 격식을 지정할 때는 Font클라스의 3개 상수 Font.PLAIN, Font.BOLD, Font.ITALIC중의 하나를 사용하여야 한다.

만일 Font객체를 사용하려면 Graphics클라스의 setFont( )메쏘드를 리용할수 있다. 즉

```
g.setFont(MyFont);
```

또한 조종부품에 서체효과를 적용하려면 조종부품의 메쏘드 setFont( )를 사용한다. 실례로 btn의 단추객체에서는 아래와 같은 명령문을 리용한다.

```
btn.setFont(MyFont);
```

이 명령문에 의해 단추의 서체가 12pt의 TimesRoman강조체로 고쳐진다.

그밖에 getFont( )메쏘드는 현재의 Graphics나 부품객체가 사용한 서체를 귀환시킨다.

실례 8-2

```
Example 8-2 AvailableFonts.java
1: import java.applet.*;
2: import java.awt.*;
3:
4: public class AvailableFonts extends Applet
5: {
6:     GraphicsEnvironment gl = GraphicsEnvironment.getLocalGraphicsEnvironment();
7:     String FontNames[ ] = gl.getAvailableFontFamilyNames( );
8:
9:     public void paint(Graphics g)
10:    {
11:        Font current, oldFont;
12:
13:        oldFont = g.getFont( );
```

```
14:         for(int i = 0; i < FontNames.length; i++)
15:         {
16:             current = new Font(FontNames[i], Font.PLAIN, 10);
17:             g.setFont(current);
18:             g.drawString(current.getName( ), 10 + i % 4 * 120, 20 + i / 4 * 15);
19:         }
20:         g.setFont(oldFont);
21:     }
22:}
```

프로그람설명

실례에서는 우선 6행에서 java.awt패키지의 GraphicsEnvironment클라스에 있는 정적메쏘드 getLocalGraphicsEnvironment( )를 리용하여 Java프로그람을 실행시키는 현재기반의 도형환경에 대한 객체 gl을 얻은 다음 7행에서 gl객체의 getAvailableFontFamilyNames( )를 사용하여 현재기반에서 사용할수 있는 모든 서체이름들을 얻어 문자렬배렬 FontNames[ ]에 귀환시킨다. 이 배렬의 매개 원소는 서체이름을 나타내는 문자렬이다. 13행은 현재객체의 기정서체를 얻고 보존한다. 14 - 19행의 순환은 문자를 사용할수 있는 서체로 설정한 다음 이 서체의 이름을 출력한다. 4개 서체가 한행을 차지한다. 16행은 Font클라스의 구성자를 사용하여 새로운 서체객체를 창조한다. 20행에서 서체를 원래의 기정값으로 복귀한다. 실례 8-2의 실행결과를 그림 8-2에서 보여주었다.

Applet Viewer: AvailableFonts.class

Applet

Arial	Arial Black	Arial Narrow	Book Antiqua
Bookman Old Style	Century	Century Gothic	Comic Sans MS
Courier New	Default	Dialog	DialogInput
Franklin Gothic Medium	Garamond	Georgia	Impact
KP CheongPong	KP CheongPongChe	KP CheonRiMa	KP CheonRiMaChe
KP KwangMyeong	KP PusKul	Lucida Bright	Lucida Console
Lucida Sans	Lucida Sans Typewriter	Lucida Sans Unicode	□□□□□□□
Microsoft Sans Serif	Monospaced	Monotype Corsiva	MS Gothic
MS Mincho	□□□□□□□□□□	MS PGothic	MS PMincho
MS UI Gothic	□□ □□□□	MSimSun	Palatino Linotype
PRK P Bukul	PRK P Chongbong	PRK P Chongbong Bold	PRK P Gothic
PRK P Myongjo	SansSerif	Serif	SimHei
SimSun	□□□□□□	Tahoma	Times New Roman
Trebuchet MS	Verdana	□□□□□□□□	□□□□□□□□□
□□□□□□□□□□□	□□□□□□□□□□□	WKFixd	WKGothic
WKLChongbong	WKLGwangmyong	WKPukgul	WKSystem

Applet started.

그림 8-2. 실례 8-2의 실행결과

8.2.3. 색갈조종

Applet에서 현시하는 문자렬이나 도형의 색갈은 Color클라스의 객체를 리용하여 조종할수 있으며 매개 Color객체는 하나의 색갈을 의미한다. 사용자는 Color클라스에서 정의한 색갈상수(붉은색, 풀색, 푸른색의 비률을 조합한 수)로 Color객체를 창조할수 있다. Color클라스는 아래의 3가지 구성자를 가진다.

```
public Color(int Red, int Green, int Blue);
public Color(float Red, float Green, float Blue)
public Color(int RGB)
```

어느 구성자를 사용하여 Color객체를 창조하든지간에 색갈에서 R(붉은색), G(풀색), B(푸른색)의 비률을 지정하여야 한다. 첫번째 구성자에서는 3개의 옹근수형파라메터를 통하여 R, G, B를 지정하며 매 파라메터가 취하는 값은 0~255사이에 있다. 두번째 구성자는 3개의 류동소수점수파라메터로 R, G, B를 지정하며 매개 파라메터의 값범위는 0~1.0사이에 있다. 3번째 구성자는 한개의 옹근수형파라메터로 RGB의 3색비률을 지정하는데 이 파라메터의 0~7bit(값범위 : 0~255)는 붉은색의 비률, 8~15bit는 풀색의 비률, 16~23bit는 푸른색의 비률을 의미한다. 실례로 아래의 명령문은 푸른색을 창조한다. 즉

```
Color blueColor = new Color(0, 0, 255);
```

Graphics객체의 setColor( )메쏘드를 호출하면 현재의 기정색갈을 새로 창조하는 색갈로 고칠수 있으며 후에 Graphics객체를 호출하여 완성하는 그리기작업에 새로 창조한 색갈을 리용할수 있다.

```
g.setColor(blueColor);
```

Color클라스가 정의한 색갈상수를 직접 사용할수도 있다. 례하면

```
g.setColor(Color.cyan);
```

Color클라스에는 모두 13개의 정적색갈상수가 정의되여있다. 여기에 black, orange, pink, grey 등이 속하며 사용할 때에는 Color를 앞붙이로 리용해야 한다.

GUI의 조종부품에 관하여 색갈과 련관있는 4개의 메쏘드가 있는데 그것들은 각각 부품의 배경색과 전경색을 설정하고 얻는데 쓰인다.

```
public void setBackground(Color c)
public Color getBackground( )
public void setForeground(Color c)
public Color getForeground( )
```

실례 8-3

```
Example 8-3 UseColor.java
1: import java.applet.Applet;
2: import java.awt.*;
3:
4: public class UseColor extends Applet
5: {
6:      Color oldColor;
7:      String[] ParamName = {"red", "green", "blue"};
8:      int[] RGBarray = new int[3];
9:
10:     public void init( )
11:     {
12:          for(int i = 0; i < ParamName.length; i++) //HTML파일이 지정하는 3색을 얻기
13:          RGBarray[i] = Integer.parseInt(getParameter(ParamName[i]));
14:     }
15:     public void paint(Graphics g)
16:     {
17:          oldColor = g.getColor( );   //원래의 기정색을 보존
18:          g.setColor(new Color(RGBarray[0], RGBarray[1], RGBarray[2]));
                                        //새로운 색을 설정
19:          g.drawString("How do you think about Current color:"
                                        + g.getColor( ).toString( ), 10, 20);
21:          g.setColor(oldColor);      //원래의 색을 복귀
22:          g.drawString("Back to old default color:" + g.getColor( ).toString( ), 10, 40);
23:     }
24:}
```

실례 8-3의 실행결과는 그림 8-3에서 보여준다.

그림 8-3. 실례 8-3의 실행결과

프로그람설명

실례 8-3은 HTML파일로부터 R, G, B의 색갈상수를 읽어들여 색갈을 창조하며 새로운 색갈을 리용하여 문자렬을 현시한다. 21행에서는 기정색갈을 복귀한다.

이 프로그람에 대응하는 HTML파일은 아래와 같다.

```
<html><head><title>UseColor</title></head>
    <body><hr>
        <applet code = UseColor width = 450 height = 250>
            <PARAM name = red value = 255>
            <PARAM name = green value = 0>
            <PARAM name = blue value = 0>
        </applet><hr>
</body></html>
```

3개 파라메터의 수값(0~255사이에 있어야 한다.)을 변경하여 서로 다른 색갈을 지정할수 있으며 Java Applet프로그람에서 다시 번역하지 않아도 된다.

8.2.4. 화상현시

화상의 자료량은 도형보다 많으므로 일반적으로 프로그람에서 자동적으로 화상을 그리지 못하고 이미 콤퓨터의 하드디스크나 홈페지에 있는 2진화상파일을 기억기에 직접 넣는 방법으로 화상을 현시한다. 화상파일은 여러가지 형식을 가질수 있다.(례: bmp파일, gif파일, tiff파일 등) 여기서 gif는 인터네트상에서 자주 쓰는 화상파일형식이다.

Java에서는 Graphics클라스의 drawImage( )메쏘드를 리용하여 화상을 현시할수 있다. 아래의 례제를 고찰하자.

실례 8-4

```
Example 8-4 DrawMyImage.java
1: import java.awt.*;
2: import java.applet.Applet;
3: public class DrawMyImage extends Applet
4: {
5:      Image myImage;
6:      public void init( )
7:      {
8:          myImage = getImage(getDocumentBase( ), "blackbrd.gif");
9:      }
10:     public void paint(Graphics g)
```

```
11:    {
12:          g.drawImage(myImage, 0, 0, this);
13:    }
14:}
```

프로그람설명

실례 8-4에서는 Image클라스의 객체 myImage를 사용하여 화상자료를 보존한다. Applet클라스의 getImage( )메쏘드는 체계가 정의한 메쏘드이며 이 메쏘드는 화상파일의 내용을 기억기의 Image객체에 적재한다. getImage( )메쏘드는 두개의 파라메터를 가진다. 첫번째 파라메터는 화상파일이 있는 URL주소로서 우의 례제에서는 화상파일을 HTML파일과 같은 경로에 보존하므로 Applet의 HTML파일을 포함하는 URL주소를 지적한다. 여기서 getDocumentBase( )메쏘드는 Applet의 메쏘드이다. 귀환값은 Applet의 HTML파일이 있는 URL주소이다. getImage( )메쏘드의 두번째 파라메터는 화상파일이름으로서 Java가 식별할수 있는 화상파일형식 bmp, gif, jpeg 등이여야 한다.

drawImage( )는 Graphics클라스에서 화상을 현시하는데 쓰이는 메쏘드이다. 첫번째 파라메터는 화상자료가 있는 Image객체이다. 두번째, 세번째 파라메터는 화상에 대한 왼쪽웃모서리점의 자리표이며 그것들은 화상이라는 용기에서 현시위치를 결정한다. 마지막 파라메터는 화상을 현시하는 용기객체이다. 실례 8-4에서의 this는 현재의 Applet객체를 의미하고있다.

8.2.5. 동화상효과실현

동화상은 Java Applet에서 사람들의 흥미를 가지는 특성의 하나이다. Java를 리용하여 동화상을 실현하는 원리는 동화상을 방영하는것과 류사하다. 즉 몇가지 련관있는 화상이나 그림들을 련속적으로 화면에 먼저 현시하고 다음에 없애는 순환반복과정으로 동화상효과를 얻을수 있다. 아래의 례제를 고찰하자.

실례 8-5

```
Example 8-5 ShowAnimator.java
1: import java.applet.Applet;
2: import java.awt.*;
3:
4: public class ShowAnimator extends Applet
5: {
6:      Image[ ] m_Images;        //도형렬을 보존하는 Image배렬
7:      int totalImages = 4;        //도형렬의 도형개수설정
```

```
8:     int currentImage = 0; //현재시각에 현시하여야 하는 도형번호
9:
10:    public void init( )
11:    {
12:        m_Images = new Image[totalImages];
13:
14:        for(int i = 0; i < totalImages; i++)
15:            m_Images[i] = getImage(getDocumentBase(), "images\\Img00" + (i + 1) + ".gif");
16:    }
17:    public void start( )
18:    {
19:        currentImage = 0;  //첫째 화상부터 시작하여 현시
20:    }
21:    public void paint(Graphics g)
22:    {
23:        g.drawImage(m_Images[currentImage], 50, 50, this);
24:        currentImage = ++currentImage % totalImages;
                       //현시하여야 할 다음도형의 번호를 계산
25:        try{
26:            Thread.sleep(500);
27:        }
28:        catch(InterruptedException e)
29:        {
30:            showStatus(e.toString( ));
31:        }
32:        repaint( );
33:    }
34:}
```

프로그람설명

실례 8-5에서는 현재의 프로그람토막처리가 일정한 시간 잠자게 하기 위해서 Thread.sleep( )메쏘드를 사용하고있다. 이때 매 그림은 다른 그림을 현시하기전에 화면상에 잠간 머무른다. 14-15행의 순환은 Applet의 getImage( )메쏘드를 사용하여 모든 .gif화상파일을 얻고있다. 21-33행의 paint( )메쏘드는 화상을 한번 현시하고 소거한 후에 다시 Image객체배렬의 다음화상을 현시한다. 그림 8-4는 실례 8-5의 실행 결과이다.

그림 8-4. 실례 8-5의 실행결과

제3절. Java의 사건처리

- 사건은 사용자대면부에서 사용자의 작용에 의하여 생성된다.
- 사건대상 례: ActionEvent, ItemEvent, WindowEvent…
- 사건원천 례: Button, MenuItem…
- 사건처리방식: 위탁사건모형

도형사용자대면부에서의 명령입력은 도형대면부요소에 마우스를 찰칵 또는 두번 찰칵하거나 건반을 리용하여 실현한다. 사용자의 명령을 충분히 접수하자면 도형사용자대면부의 체계가 우선 마우스와 건반의 조작들을 식별하고 상응한 응답을 할수 있어야 한다. 보통 건반이나 마우스의 조작은 체계가 정의한 사건을 일으킬수 있으며 사용자프로그람에서는 매개 구체적인 사건발생을 정의하여 이에 대해 응답한다. 이 코드들은 그것의 사건이 발생할 때 체계에 의하여 자동호출된다. 이것이 도형사용자대면부에서 사건과 사건응답의 기본원리이다.

Java의 사건처리기구에서는 위탁사건모형(그림 8-5)을 도입하여 서로 다른 사건이 서로 다른 감시자에 의하여 처리되게 한다.

도형사용자대면부에서 사건을 발생시킬수 있는 매개 부품을 **사건원천**이라고 하며 서로 다른 사건원천에서 발생한 사건의 종류는 서로 다르다.

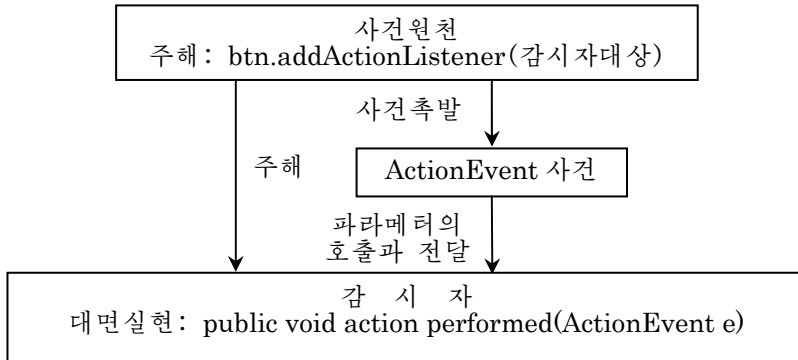



그림 8-5. 위탁사건모형

실례로 Button객체나 MenuItem객체 등은 사건원천으로서 ActionEvent클라스가 나타내는 사건 ACTION_PERFORMED를 일으킬수 있다. Checkbox객체 등은 사건원천으로서 ItemEvent클라스의 ITEM_STATE_CHANGES사건을 일으킬수 있다.

사건원천에서 발생하는 사건이 프로그람에 의해 처리되자면 이 사건을 처리할수 있는 감시자에게 사건원천을 등록해야 한다. 실례로 Button객체는 ActionListener대면을 실현하는 객체에 등록된다 . 때문에 이 객체만이 Button객체에서 발생한 ActionEvent클라스의 사건을 능히 처리할수 있다. 이때 감시자는 사건원천을 포함하는 용기일수도 있고 그밖의 객체일수도 있다. 구체적인 등록메쏘드는 사건원천자체의 메쏘드를 호출하는것(실례로 Button클라스자체의 addActionListener( )메쏘드를 호출하는것)이며 여기서 실제파라메터가 감시자객체로 되여 실현된다. 감시자는 련관있는 대면을 실현하므로 대면을 실현하는 모든 추상메쏘드에 대하여 구체적인 메쏘드본체를 써주어야 하며 사건원천에서 발생하는 사건들에 대응하는 코드처리는 이 메쏘드본체에서 작성된다.

실례로 Button에서 발생한 사건에 대한 코드처리는 Button객체가 등록된 감시자의 actionPerformed( )메쏘드에서 작성되여야 한다. 이 메쏘드는 ActionListener대면에서 같은 이름의 추상메쏘드에 대한 구체적인 실현이다. 사건원천에서 감시자가 처리할수 있는 사건을 발생시킬 때 사건원천은 이 사건을 실제파라메터로 감시자에게 보내여 이 사건의 메쏘드를 책임적으로 처리한다.(즉 위탁처리한다) 여기서 감시자는 꼭 사건원천을 가지는 용기객체가 되여야 하는것은 아니다. 이렇게 하여 처리는 프로그람의 사건처리코드와 GUI대면부구성코드를 분리한다. 이것은 프로그람구조의 최량화에 유리하다. Java의 모든 사건클라스와 사건을 처리하는 감시자대면은 java.awt.event패키지에서 정의된다. 사건클라스의 계층구조를 그림 8-6에 보여주었다.

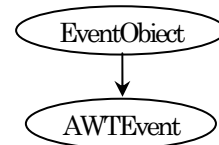



그림 8-6. AWTEvent클라스체계구조도

이 구조도에 포함된 사건클라스는 많다. 그것들은 java.awt.AWTEvent클라스의 하위클라

스들이고 java.awt.AWTEvent클라스는 java.util.EventObject클라스의 하위클라스이다. EventObject는 앞에서 이미 사용한적이 있는 중요한 메쏘드인 getSource( )를 가지고있다. 이 메쏘드는 사건을 발생시키는 사건원천을 귀환시키며 거의 모든 사건클라스들은 이 메쏘드를 사용한다. 주의해야 할것은 매개 사건클라스는 단지 하나의 사건에만 대응하는것이 아니라는것이다. 실례로 KeyEvent클라스는 KEY_PRESSED(건누르기), KEY_RELEASED(건해방), KEY_TYPED(건치기)의 3개의 구체적인 사건에 대응할수 있다. KeyEvent클라스의 객체가 과연 어느 사건을 나타내는가 하는것은 getID( )메쏘드를 호출하여 얻은 메쏘드의 귀환값을 KEY_PRESSED 등 몇개의 상수와 비교하여 알수 있다. 매개 사건클라스의 객체는 getID( )메쏘드를 가지며 그것들은 상위클라스 AWTEvent로부터 계승된다.

java.awt.event패키지에는 11개의 감시자대면이 정의되여있다. 그리고 매개 대면은 련관된 사건을 처리하는 몇개의 추상메쏘드들을 포함하고있다. 일반적으로 매개 사건클라스들은 그에 대응한 감시자를 가지며 사건클라스에서의 매개 구체적인 사건류형은 그에 대응하는 구체적인 추상메쏘드를 가진다. 구체적인 사건이 발생할 때 이 사건은 사건클라스의 객체를 실제파라메터로 하여 내장한 다음 대응한 구체적인 메쏘드에 전달하며 이 구체적인 메쏘드는 발생한 사건에 응답하고 처리를 진행한다. 실례로 ActionEvent클라스사건에 대응하는 대면은 ActionListener이며 이 대면에서는 다음의 추상메쏘드를 정의하고있다.

```
public void actionPerformed(ActionEvent e);
```

ActionEvent사건을 처리해야 할 클라스들이 ActionListener대면을 실현하려면 반드시 우의 actionPerformed( )메쏘드를 재정의하여야 하며 재정의메쏘드본체에서 보통 파라메터 e와 련관있는 메쏘드를 호출하여야 한다. 실례로 e.getSource를 호출하여 ActionEvent사건을 발생시키는 사건원천을 찾고 다음에 해당한 조치를 취하여 이 사건을 처리한다.

제4절. 표식자, 단추와 그의 동작사건

- Label은 읽기전용의 부품이다.
- Label의 구성자: Label( ), Label(String text),
  Label(String text, int alignment)
- Button은 마우스를 찰칵할 때 사건을 발생한다.
- ActionListener대면이 사건처리에 참가한다.
- 구성자:Button( ),Button(String text)

8.4.1. 표식자

표식자(Label)는 사용자의 입력은 접수할수 없고 그 내용을 볼수만 있는 본문현시구역으로서 정보설명의 역할을 한다. 매개 표식자는 Label클라스를 리용하여 표시한다.

1) 표식자창조

표식자객체를 창조할 때 이 표식자의 현시문자렬을 작성해야 한다.

```
Label prompt = new Label("하나의 옹근수를 입력하십시오:");
```

2) 상용메쏘드

만일 표식자에 현시한 본문을 수정하려면 Label객체의 setText(String s)메쏘드를 사용할수 있다. 마찬가지로 Label객체의 getText( )메쏘드를 호출하여 그의 내용을 얻을수도 있다. 아래의 코드부분에서는 표식자의 본문내용을 수정한다.

```
if(prompt.getText( ) == "안녕하십니까")
    prompt.setText("안녕히 계십시오");
else if(prompt.getText( ) == "안녕히 계십시오")
    prompt.setText("안녕하십니까");
```

3) 사건생성

표식자는 사용자의 입력을 접수할수 없으므로 사건을 일으킬수 없다. 즉 표식자는 사건원천이 아니다.

8.4.2. 단추

단추(Button)는 도형사용자대면부에서 아주 중요한 기본부품이다. 사용자가 단추를 찰칵할 때 체계는 이 단추와 련관있는 프로그람을 자동실행한다.

1) 창조

단추창조시 아래의 명령문을 리용한다. 구성자에서 문자렬파라메터는 단추의 표식자를 지정하고있다.

```
Button enter = new Button("조작");
```

2) 상용메쏘드

단추의 getLabel( )메쏘드를 호출하면 단추의 표식자문자렬을 얻을수 있다. 단추의 setLabel(String s)메쏘드를 호출하면 단추의 표식자를 문자렬 s로 설정할수 있다.

3) 사건생성

사용자가 단추를 찰칵할 때 하나의 동작사건을 일으킨다. 단추가 일으키는 동작사건에 응답하는 프로그람에서는 반드시 단추를 ActionListener대면을 실현하고있는 동작사건감시자에게 등록하여야 하며 동시에 이 대면의 actionPerformed (ActionEvent e)메쏘드본체를 작성하여야 한다. 메쏘드본체에서 e.getSource( )메쏘드를 호출하여 동작사건을 일으키는 단추객체인용을 얻을수 있으며 e.getActionCommand( )메쏘드를 호출하여 단추의 표식자나 사전에 이 단추가 설정한 지령행을 얻을수 있다.

실례 8-6

```
Example 8-6 BtnLabelAction.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class BtnLabelAction extends Applet implements ActionListener
6: {
7:      Label prompt;
8:      Button btn;
9:      public void init( )
10:     {
11:         prompt = new Label("안녕하십니까");
12:         btn = new Button("조작");
13:         add(prompt);
14:         add(btn);
15:         btn.addActionListener(this);
16:     }
17:     public void actionPerformed(ActionEvent e)
18:     {
```

```
19:         if(e.getSource( ) == btn)
20:             if(prompt.getText( ) == "안녕하십니까")
21:                 prompt.setText("다시 만납시다");
22:             else
23:                 prompt.setText("안녕하십니까");
24:     }
25:}
```

프로그람설명

실례 8-6에서는 표식자객체 prompt와 단추객체 btn의 사용을 보여준다. 5행에서 정의한 클라스는 ActionListener대면을 실현하며 이것이 ActionEvent사건의 감시자이다. 15행은 단추객체 btn을 이 감시자에게 등록한다. 이렇게 하여 그것은 btn이 일으키는 동작사건을 감시하고 처리한다. 17-24행의 actionPerformed( )메쏘드는 사용자가 btn을 찰칵할 때 체계에 의하여 자동호출된다. 19행은 동작사건이 단추에 의하여 일어나는가를 판단하며 일어나면 prompt객체의 본문표식자를 수정한다. 그림 8-7은 실례 8-6의 실행결과이다.

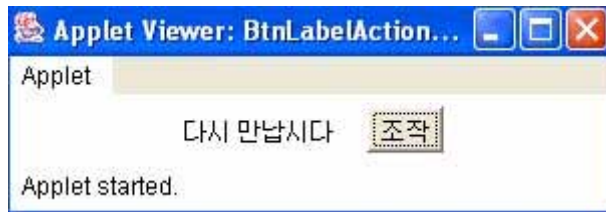



그림 8-7. 실례 8-6의 실행결과

8.4.3. 동작사건

ActionEvent클라스는 한개의 사건만을 포함한다. 즉 ACTION_PERFORMED동작사건을 집행한다. ACTION_PERFORMED는 어떤 동작을 일으켜서 실행하는 사건이다.

이 사건을 촉발시킬수 있는 동작은 다음과 같다.

- 단추찰칵
- 목록에서 선택항목의 두번찰칵
- 차림표항목의 선택
- 본문칸에서의 입력되돌리기

ActionEvent클라스의 중요메쏘드는 다음과 같다.

1) public String getActionCommand( )

이 메쏘드는 사건을 일으키는 동작의 명령이름을 귀환시킨다. 이 명령이름은 setActionCommand( )메쏘드를 호출하여 사건원천부품에 지정할수도 있고 사건원천의 기정명령이름을 사용할수도 있다. 실례로 단추부품 m_Button은 ACTION_PERFORMED사건의 사건원천이며 아래의 명령문은 이 단추객체의 동작명령이름을 《명령이름》으로 생성하고 그것을 현재의 감시자에게 등록한다.

```
Button m_Button = new Button("단추표식자");
m_Button.setActionCommand("명령이름");
m_Button.addActionListener(this);
```

동작사건의 감시자는 동작을 실현하여야 하는데 그 메쏘드는 아래와 같다.

```
public void actionPerformed(ActionEvent e)
{
    if(e.getActionCommand( ) == "명령이름")
    …
}
```

여기서 setActionCommand( )메쏘드와 getActionCommand( )메쏘드가 서로 다른 클라스에 속한다는데 주의하여야 한다. getActionCommand( )메쏘드는 ActionEvent클라스의 메쏘드이지만 setActionCommand( )메쏘드는 동작사건을 발생시키는 사건원천의 메쏘드(실례로 단추, 차림표항목 등의 메쏘드)이다. 사건원천객체 역시 setActionCommand( )메쏘드를 전문적으로 호출하지 않고 명령이름을 지정할수 있다. 이때 getActionCommand( )메쏘드는 기정인 명령이름을 귀환시킨다. 실례로 동작명령이름을 설정하는 한개 명령문을 빼버리면 감시자대면의 메쏘드는 아래와 같이 작성할수 있다.

```
public void actionPeformed(ActionEvent e)
{
    if(e.getActionCommand( )== "단추표식자")
    …
}
```

보는바와 같이 단추의 기정이름은 《단추표식자》이다. getActionCommand( )메쏘드를 호출하면 동작명령을 생성하는 서로 다른 사건원천을 구분할수 있으며 actionPerformed( )메쏘드는 서로 다른 사건원천이 일으키는 사건에 대하여 기다림처리를 구분하게 된다

2) public int getModifiers( )

만일 동작사건을 발생시키는 동시에 사용자가 Ctrl이나 Shift와 같은 기능건을 눌렀다면 이 사건의 getModifiers( )메쏘드를 호출하여 이 기능건들을 얻고 구분할수 있

다. 실제상 동작사건을 다시 몇개의 사건으로 세분하여 한개의 명령을 몇개의 명령들로 나눌수 있다. getModifiers( )메쏘드의 귀환값을 ActionEvent클라스의 정적상수 ALT_MASK, CTRL_MASK, SHIFT_MASK, META_MASK와 비교하면 사용자가 어느 기능건을 눌렀는가를 판단할수 있다.

제5절. 본문마당, 본문구역과 본문사건

- 본문부품은 문자렬을 편집할수 있는 부품이다.
  TextField, TextArea
- 비밀열쇠입력시 사용하는 메쏘드
  setEchoChar(char c), getEchoChar( )
- TextArea는 여러행 문자렬의 편집이 가능하다.

8.5.1. 본문사건

TextEvent클라스는 한개의 사건만을 포함한다. TEXT_VALUE_CHANGED로 본문구역의 본문내용을 변경한다.(례: 문자삭제, 문자건입력) 이 사건은 비교적 간단하며 사건류형의 메쏘드와 상수를 특별히 판단하지 않아도 된다.

8.5.2. 본문마당과 본문구역

Java에서 본문처리에 리용되는 기본부품에는 2가지가 있다. 즉 단일행본문마당 **TextField**와 여러행본문구역 **TextArea**가 있는데 모두 TextComponent의 하위클라스이다.

1) 창조

본문부품을 창조하는 동시에 본문부품의 초기본문문자렬을 표시할수 있다. 례를 들어 아래의 명령문은 10행, 45렬의 여러행본문구역을 창조하고있다.

```
TextArea textArea1 = new TextArea(10, 45);
```

또한 8개문자를 허용하며 초기문자렬이 카드번호인 단일행본문마당은 아래의 명령문을 사용하여 창조할수 있다.

```
TextField name = new TextField("카드번호", 8);
```

2) 상용메쏘드

사용자는 이미 창조한 본문구역에서 본문정보를 자유로 입력하고 편집할수 있으며 사용자가 입력한 정보는 TextComponent의 getText( )메쏘드를 호출하여 얻을수

있다. 이 메쏘드의 귀환값은 문자렬이다.

만일 프로그람에서 본문구역에 현시할 내용을 값주기하려면 TextComponent의 setText()메쏘드를 리용할수 있다. 실례로 아래의 명령문

```
textArea1.setText("안녕하십니까, 환영합니다!");
```

는 본문구역의 내용을 《안녕하십니까, 환영합니다!》로 설정한다.

어떤 경우에는 본문구역을 편집할수 없게 설정하여야 할 경우가 있다. 실례로 전화카드의 카드번호는 체계가 자동적으로 생성하고 사용자가 마음대로 변동시킬수 없다. 이때 아래의 명령문을 리용하여 전화카드번호에 대응하는 본문칸 cardNo를 사용자가 대면을 통하여 변동할수 없게 할수 있다. 즉

```
CardNo.setEditable(false);
```

그 밖에 isEditable()메쏘드를 호출하여 현재의 본문구역을 편집할수 있는가를 판단할수 있다.

TextComponent에는 또한 본문구역에서 본문의 상태를 지정하거나 얻는데 쓰이는 메쏘드가 있다. select(int start, int end)메쏘드는 지정한 시작끝위치에 근거하여 본문을 선택한다. selectAll( )메쏘드는 본문구역에서의 모든 본문을 선택하며 setSelectionStart( )메쏘드와 setSelectionEnd( )메쏘드는 각각 본문을 선택하는 시작과 끝위치를 지정한다. 만일 본문구역에서 이미 선택한 본문이 있다면 getSelectionStart( )메쏘드와 getSelectionEnd( )메쏘드를 리용하여 선택한 본문의 시작, 끝위치를 얻을수 있다. 선택한 본문을 얻으려면 getSelectedText( )메쏘드를 호출할수 있다.

TextComponent클라스를 계승하는 메쏘드외에 TextField는 자기의 특수한 메쏘드를 정의하고있다. 실례로 본문구역의 내용을 화면에 보여주지 말아야 하는 경우(례: 비밀열쇠입력시) 아래의 메쏘드를 적용할수 있다.

```
TextField tf = new TextField("비밀열쇠입력");
tf.setEchoChar('*');
```

이렇게 하면 TextField에서의 매개 문자(동양문자이든 서양문자이든)는 한개의 별표(*)로 반영되며 따라서 실제문자를 볼수 없게 된다. 그밖에 TextField는 echoCharIsSet( )메쏘드를 정의하여 현재 본문칸이 무반영상태에 놓여있는가를 확인한다. getEchoChar()메쏘드는 현재 본문칸에서 무반영상태의 문자를 얻는다.

TextArea도 TextComponent클라스를 계승하는 메쏘드외에 2개의 특수한 메쏘드 append(String s)와 insert(String s, int index)를 정의하고있다.

append(String s)메쏘드는 현재 본문구역에 이미 있는 본문의 뒤에 문자렬파라메터 s가 지정한 본문내용을 추가한다. insert(String s, int index)메쏘드는 문자렬 s를 이미 있는 본문의 지정한 위치에 삽입한다.

3) 사건응답

TextField와 TextArea의 사건응답은 그것들의 상위클라스 TextComponent에 의해 결정되므로 먼저 TextComponent의 사건응답을 고찰한다. TextComponent는 하나의 사건을 일으킬수 있다. 즉 사용자가 본문구역의 본문을 수정할 때 례를 들어 본문의 추가, 수정 등의 조작을 할 때 TextEvent객체가 나타내는 본문변경사건을 일으킨다. 여기서 TextField는 TextArea에 비하여 한개의 사건을 더 생성하여 사용자본문칸에서 입력건을 누를 때 동작사건을 나타내는 ActionEvent사건을 일으킨다. 반대로 TextArea는 ActionEvent사건을 생성할수 없고 addActionListener( )의 메쏘드도 가지지 않는다.

만일 우의 두 사건에 응답하려면 본문칸을 TextListener대면을 실현하고있는 본문에 가입시켜 사건감시자를 변경시키고 다음에 ActionListener대면을 실현하는 동작사건감시자를 변경시켜야 한다.

```
textField1.addTextListener(this);
textField1.addActionListener(this);
```

감시자내부에서 각기 본문응답을 정의하며 사건들사이의 동작을 변경시켜야 한다.

```
public void textValueChanged(TextEvent e);
public void actionPerformed(ActionEvent e);
```

이것은 본문칸이 일으키는 본문에 응답하여 사건과 동작사건을 변경시킬수 있다. 본문변경사건에 대한 메쏘드 e.getSource( )를 호출하면 이 사건을 일으키는 본문칸의 객체인용을 얻을수 있으며 이 본문칸의 getText( )메쏘드를 호출하여 변경후의 본문내용을 얻을수 있다.

```
String afterChange = ((TextField)e.getSource( )).getText( );
```

실례 8-7

```
Example 8-7 TextComponentEvent.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TextComponentEvent extends Applet
6:                                     implements TextListener, ActionListener
7: {
8:      TextField tf;
9:      TextArea ta;
10:     public void init( )
11:     {
12:          tf = new TextField(45);
13:          ta = new TextArea(10, 45);
14:          add(tf);
15:          add(ta);
```

```
16:         tf.addActionListener(this);
17:         tf.addTextListener(this);
18:     }
19:     public void textValueChanged(TextEvent e)
20:     {
21:         if(e.getSource( ) == tf)
22:             ta.setText(((TextField)e.getSource( )).getText( ));
23:     }
24:     public void actionPerformed(ActionEvent e)
25:     {
26:         if(e.getSource( ) == tf)
27:             ta.setText(" ");
28:     }
29:}
```

프로그람설명

실례 8-7의 Applet프로그람은 TextField객체 tf와 TextArea객체 ta를 포함하고있다. 이 클라스는 TextListener대면과 ActionListener대면을 실현하는 본문사건과 동작사건의 감시자이다. 16, 17행에서 tf객체를 두개의 감시자에 각각 등록한다. 19-23행에서 정의한 textValueChanged( )메쏘드는 본문사건을 처리하며 사용자가 tf에서 본문을 입력하거나 수정할 때 ta본문구역에서 동기적인 복사를 얻을수 있다. 24-28행이 정의한 actionperformed( )메쏘드는 동작사건을 처리하는데 쓰이며 사용자가 tf에서 되돌이건을 누를 때 ta의 본문을 지워버린다. 그림 8-8은 실례 8-7의 실행결과이다.

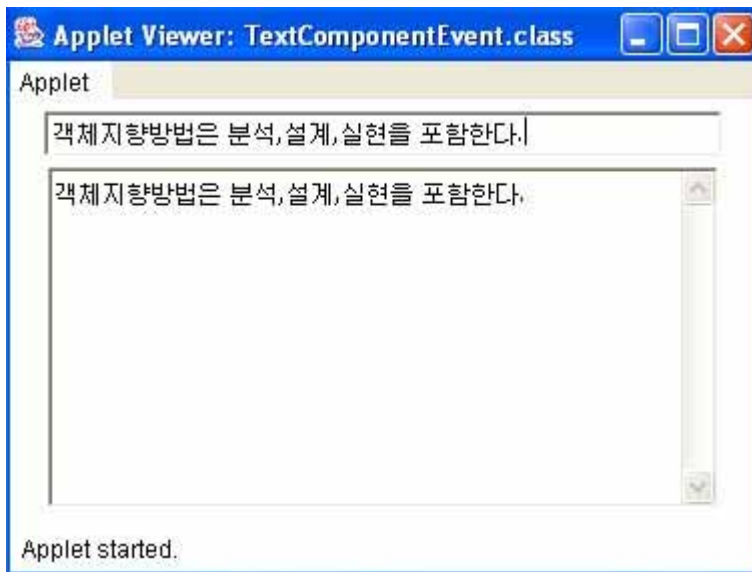



그림 8-8. 실례 8-7의 실행결과

제6절. 단일선택단추, 검사칸, 목록과 사건선택

- Checkbox, CheckboxGroup, Choice, List부품의 사건감시자는 ItemListener이다.
- Checkbox는 on과 off상태로 선택여부를 결정한다.
- CheckboxGroup은 단일선택단추와 같이 여러개중 하나만 선택한다.
- Choice는 내리펼구기목록으로서 Combobox와 같다.
- List 역시 내리펼구기목록이며 여러 항목을 선택할수 있다.

8.6.1. 사건선택

ItemEvent(사건선택)클라스는 하나의 사건만을 포함한다. 즉 선택항목의 선택상태가 변경되는 사건 ITEM_STATE_CHANGED를 나타낸다. 이 사건을 일으키는 동작은 다음과 같다.

· 목록클라스 List객체에 대하여 선택항목의 선정이나 비선정상태를 변경한다.

· 내리펼구기목록클라스 Choice객체에 대하여 선택항목의 선정이나 비선정상태를 변경한다.

· 검사칸클라스 Checkbox객체의 선정이나 비선정상태를 변경한다.

· 검사칸차림표항목 CheckboxMenuItem객체의 선정이나 비선정상태를 변경한다.

ItemEvent클라스의 주요메쏘드는 다음과 같다.

1) public ItemSelectable getItemSelectable()

이 메쏘드는 선택상태의 변경을 일으키는 사건원천을 귀환시킨다. 실례로 항목선택변경의 List객체나 선택상태변경의 Checkbox객체들은 선택상태변경사건을 일으킬수 있는데 모두 ItemSelectable대면을 실현하고있는 클라스의 객체이다. getItemSelectable()메쏘드가 귀환시키는것은 이 클라스들의 객체인용이다.

2) public Object getItem()

이 메쏘드는 선택상태변경사건의 구체적인 선택항목을 귀환시킨다. 실례로 Choice클라스에서 이 메쏘드를 호출하면 사용자가 어느 선택항목을 선정하였는가를 알수 있다.

3) public int getStateChange()

이 메쏘드는 구체적인 선택상태변경의 류형을 귀환시키며 그의 귀환값은 ItemEvent클라스가 렬거하는 몇개의 정적상수들중의 하나이다.

ItemEvent.SELECTED: 선택항목이 선택되였다는것을 의미한다.

ItemEvent.DESELECTED: 선택항목을 포기하고 선택하지 않았음을 의미한다.

8.6.2. 검사칸

1) 창조

검사칸은 Checkbox클라스의 객체를 리용하여 표시한다. 검사칸객체의 창조시 그의 본문설명표식자를 동시에 지정할수 있으며 이 본문표식자는 간단히 검사칸의 의미와 작용을 설명하는 내용으로 지정한다.

```
Checkbox backg = new Checkbox("배경색");
```

2) 상용메쏘드

매개 검사칸은 오직 2가지 상태만을 가진다. 즉 check상태와 uncheck상태이다. 임의의 시각에 검사칸은 오직 이 두개의 상태중 하나의 상태에 놓인다. 사용자는 검사칸이 선택되였는가를 알아보는데 Checkbox의 getState( )메쏘드를 리용하는데 이 메쏘드의 귀환값은 론리형이다. 만일 검사칸이 선택되였으면 true를 귀환시키며 그렇지 않으면 false를 귀환시킨다. Checkbox의 setState( )메쏘드를 호출하여 검사칸의 선택을 설정할수 있다. 실례로 아래의 명령문은 Checkbox가 선택상태에 있게 한다.

```
backg.setState(true);
```

3) 사건응답

사용자가 검사칸을 찰칵하여 그 선택상태가 변경될 때 ItemEvent클라스가 나타내는 선택사건을 일으킬수 있다. 만일 이 검사칸이 아래의 명령문을 리용하였다면 즉

```
backg.addItemListener(this);
```

는 그 자체를 ItemEvent사건의 감시자 ItemListener에게 등록한다는것을 의미한다. 이 경우 체계는 이 ItemListener에서의 메쏘드를 자동호출할수 있다.

```
public void itemStateChanged(ItemEvent e);
```

이것은 검사칸의 상태변경에 응답한다. 그러므로 실제적으로 ItemListener대면의 감시자를 실현하고있다. 검사칸을 받아들인 용기는 이 메쏘드를 구체적으로 실현하여야 한다. 이 메쏘드의 본체는 보통 이러한 명령문을 포괄한다. 선택사건을 호출하는 e.getItemSelectable( )메쏘드는 선택사건을 일으키는 사건원천객체인용을 얻을수 있으며 e.getState( )메쏘드는 선택사건직후의 상태를 얻을수 있다.

또한 사건원천객체자체의 메쏘드를 직접 리용하여 조작을 진행할수 있다. 주의해야 할것은 getItemSelectable( )메쏘드의 귀환값이 Selectable대면을 실현한 객체이라는것이다. 그러므로 사건원천객체류형으로 변환하여야 한다. 실례로

```
((Checkbox)e.getItemSelectable( )).getState( );
((Checkbox)e.getItemSelectable( )).setState(false);
```

실례 8-8

```
Example 8-8 TextCheckbox.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestCheckbox extends Applet implements ItemListener
6: {
7:     Checkbox ckb;
8:     Button btn;
9:     public void init( )
10:    {
11:        ckb = new Checkbox("배경색");
12:        btn = new Button("효과");
13:        add(ckb);
14:        add(btn);
15:        ckb.addItemListener(this);
16:    }
17:    public void itemStateChanged(ItemEvent e)
18:    {
19:        Checkbox temp;
20:        if(e.getItemSelectable( ) instanceof Checkbox)
21:        {
22:            temp = (Checkbox)(e.getItemSelectable( ));
23:            if(temp.getState( ))
24:                btn.setBackground(Color.cyan);
25:            else
26:                btn.setBackground(Color.gray);
27:        }
28:    }
29:}
```

프로그람설명

실례 8-8의 프로그람은 ItemListener대면을 실현한 Applet를 정의하고있다. 여기에 한개의 검사칸과 한개의 단추를 포함한다. 11행에서 창조한 검사칸의 표식자를 《배경색》으로 설정해주며 15행은 그것을 선택사건의 감시자에게 등록한다. 17-28행에서 정의한 itemStateChanged( )메쏘드는 감시를 받는 선택사건을 구체적으로 처리하는데 쓰인다. 20행은 선택사건을 일으키는 사건원천이 Checkbox객체인가를 검사하

며 객체이면 이 사건원천의 인용을 객체 temp에 값주기한다. 23행은 검사칸의 표식자가 《배경색》인가를 검사한다. 23-26행은 if/else구조를 리용하여 검사칸이 선택상태에 있는가 없는가에 따라 단추 btn의 배경색을 설정한다.

그림 8-9는 실례 8-8의 실행결과이다.

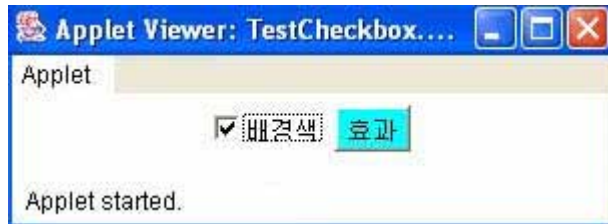



그림 8-9. 실례 8-8의 실행결과

8.6.3. 검사칸그룹

1) 창조

Checkbox는 둘중에 하나를 선택하는 경우에 리용한다. 여러개중 하나를 선택하는데 검사칸그룹을 리용할수 있다. 검사칸그룹은 Checkbox의 모임으로서 CheckboxGroup클라스의 객체를 리용하여 표시한다. 아래의 명령문은 서로 다른 서체를 가지는 3개의 검사칸그룹을 창조한다.

```
style = new CheckboxGroup( );
p = new Checkbox("보통", true, style);
b = new Checkbox("고직체", false, style);
i = new Checkbox("사선체", false, style);
```

CheckboxGroup을 용기에 추가할 때 매개 검사칸을 용기에 하나하나 추가하여야 하며 Checkboxgroup객체를 사용하여 한번에 추가할수 없다. 우의 검사칸들을 추가하자면 아래의 명령문을 인용하여야 한다.

```
add(p);
add(b);
add(i);
```

2) 상용메쏘드

검사칸의 선택은 서로 반대이다. 즉 사용자가 배렬의 1개의 단추를 선정하면 다른 단추는 비선택상태에 자동적으로 놓이게 된다. CheckboxGroup의 getSelectedCheckbox( )메쏘드를 호출하면 사용자가 어느 단추를 선택하였는가를 알수 있으며 이 메쏘드는 사용자가 선택한 Checkbox객체를 귀환시킨다. 이 객체의 getLabel( )메쏘드를 호출하면 사용자가 무슨 정보를 선택하였는가를 알수 있다. 마찬가지로 CheckboxGroup의 setSelectedCheckbox( )메쏘드를 호출하여 프로그람에서 검사칸그룹의 단추를 지정할수 있다. 아래의 명령문은 서체의 형식을 사선체로 설정한다.

```
sex.setSelectedCheckbox(i);
```

또한 단추배렬의 Checkbox검사칸의 메쏘드를 직접 사용할수도 있다. 실례로

```
i.getState( );
```

를 직접 호출하면 이 단추가 선택되였는가를 알수 있다. 만일 이 단추가 선택되였다면 다른 단추는 반드시 비선택상태에 있게 된다.

3) 사건응답

CheckboxGroup클라스는 java.awt.*패키지의 클라스가 아니라 Object의 직접적인 하위클라스이다. 그러므로 단추를 찰칵하여 사건에 응답할수 없다. 그러나 단추배렬의 매개 단추를 찰칵하면 ItemEvent클라스의 사건에 응답할수 있다. 검사칸그룹에서의 매개 검사칸은 모두 Checkbox객체이며 그것들의 사건에 대한 응답과 검사칸의 사건에 대한 응답은 같다.

실례 8-9

```
Example 8-9 TestCheckboxGroup.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestCheckboxGroup extends Applet implements ItemListener
6: {
7:      CheckboxGroup style;
8:      Checkbox p, b, i;
9:      Button btn;
10:     public void init( )
11:     {
12:          style = new CheckboxGroup( );
13:          p = new Checkbox("보통", true, style);
14:          b = new Checkbox("굵은체", false, style);
15:          i = new Checkbox("사선체", false, style);
16:          btn = new Button("효과");
17:          add(p);
18:          add(b);
19:          add(i);
20:          add(btn);
21:          p.addItemListener(this);
22:          b.addItemListener(this);
23:          i.addItemListener(this);
24:     }
```

```
25:     public void itemStateChanged(ItemEvent e)
26:     {
27:         Checkbox temp;
28:         Font oldF = btn.getFont( );
29:         if(e.getItemSelectable( ) instanceof Checkbox)
30:         {
31:             temp = (Checkbox)(e.getItemSelectable( ));
32:             if(temp.getLabel( ) == "보통")
33:                 btn.setFont(new Font(oldF.getName( ), Font.PLAIN, oldF.getSize( )));
34:             if(temp.getLabel( ) == "굵은체")
35:                 btn.setFont(new Font(oldF.getName( ), Font.BOLD, oldF.getSize( )));
36:             if(temp.getLabel( ) == "사선체")
37:                 btn.setFont(new Font(oldF.getName( ), Font.ITALIC, oldF.getSize( )));
38:         }
39:     }
40:}
```

프로그람설명

실례 8-9에서는 ItemListener대면을 실현한 Applet를 정의하고있으며 여기에는 한개의 검사칸그룹과 몇개의 검사칸객체, 한개의 단추가 포함되여있다. 12행에서 창조한 검사칸그룹객체 style은 사용자입력서체의 서로 다른 격식을 지정하는데 쓰인다. 13-15행에서 창조한 검사칸객체의 서체는 각각 《보통》, 《강조체》, 《사선체》이다. 17-19행은 이 3개의 객체단추를 style검사칸그룹에 추가한다. 검사칸그룹의 매개 검사칸들은 서로 배반이며 여기서 어느 하나를 선택하면 다른 단추는 자동적으로 선택되지 않는 상태에 있게 된다. 21-23행은 이 3개의 검사칸을 ItemListener대면을 실현한 감시자에게 등록한다. 25-39행에서 정의한 itemStateChanged메쏘드는 선택사건을 처리하는데 쓰인다. 먼저 29행에서 선택사건을 일으키는 사건원천이 검사칸그룹안의 검사칸인가를 판단하고 다음에 배렬에서 매개 검사칸의 선택상태를 알아본다. 선택이면 단추의 서체를 이 검사칸이 지정한 서체형식으로 설정한다.

그림 8-10은 실례 8-9의 실행결과이다.

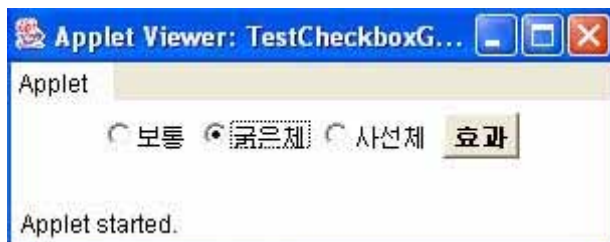



그림 8-10. 실례 8-9의 실행결과

8.6.4. 내리떨구기목록

1) 창조

내리떨구기목록(Choice) 역시 여러개중에서 하나를 선택하는 입력대면부이다. 그러나 이것은 검사칸그룹이 검사칸들을 리용하여 모든 선택항목을 렬거하는 방법과는 다르다. 내리떨구기목록의 모든 선택항목은 접기에 의하여 보이지 않기때문에 제일 앞의것 또는 사용자가 선정한것중의 하나만이 현시된다. 만일 다른 항목을 선택하려면 내리떨구기목록의 오른쪽의 내리삼각형단추를 찰칵하여 모든 선택항목을 라렬한 4각형구역을 내리떨구기해야 한다.

내리떨구기목록을 창조하는데는 선택항목을 창조하고 추가하는 두 단계가 있다.

```
Choice size = new Choice( ); //내리떨구기목록창조
Size.add("10"); //내리떨구기목록에 선택항목추가
Size.add("14");
Size.add("18");
```

2) 상용메쏘드

내리떨구기목록의 상용메쏘드에는 선택항목을 얻는 메쏘드, 선택항목을 설정하는 메쏘드, 선택항목을 추가, 제거하는 메쏘드가 있다.

getSelectedIndex( )메쏘드는 선택된 선택항목의 번호(내리떨구기목록에서 첫번째 선택항목의 번호는 0, 두번째 선택항목의 번호는 1 …)를 귀환시킨다. getSelectedItem( )메쏘드는 선정된 항목의 표식본문문자렬을 귀환시킨다. select(int index)메쏘드와 select(String item)메쏘드는 각각 프로그람에서 지정한 번호나 본문내용의 항목을 선택하게 한다. add(String item)메쏘드와 insert(String item, int index)메쏘드는 각각 새로운 항목 item을 현재 내리떨구기목록의 마지막 혹은 지정한 번호위치에 추가한다.

remove(int index)메쏘드와 remove(String item)메쏘드는 번호나 표식본문이 지정하는 항목을 내리떨구기목록으로부터 삭제한다. removeAll( )메쏘드는 내리떨구기목록의 모든 항목들을 삭제한다.

3) 사건응답

내리떨구기목록은 ItemEvent가 나타내는 선택사건을 발생시킬수 있다. 만일 항목을 ItemListener대면을 실현한 감시자 size.addItemListener에 등록하면 사용자가 내리떨구기목록의 어떤 항목을 한번 찰칵하여 선택할 때 체계는 ItemEvent클라스의 객체를 자동적으로 생성하여 이 사건의 련관정보를 포함시킨 다음 이 객체를 실제파라메터로 하여 자동적으로 호출된 감시자의 선택사건응답메쏘드에 전달한다.

```
public void itemStateChanged(Item Event e);
```

의 메쏘드에서 e.getItemSelectable( )을 호출하면 현재 선택사건을 일으키는 내리떨구기목록사건원천을 얻을수 있고 또한 내리떨구기목록의 련관메쏘드를 호출하면 사용자가 구체적으로 어느 항목을 선택하였는가를 알아낼수 있다.

```
String selectedItem=((Choice)e.getItemSelectable( )).getSelectedItem( );
```

e. getItemSelectable( )메쏘드의 귀환값에 대하여 강제형변환을 진행하여야 하며 Choice클라스의 객체인용으로 변환한 후에 Choice클라스의 메쏘드를 호출할수 있다.

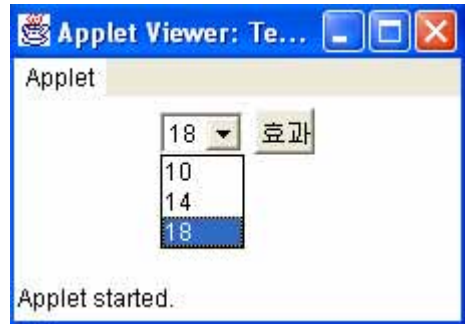
실례 8-10

```
Example 8-10 TestChoice.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestChoice extends Applet implements ItemListener
6: {
7:     Choice size;
8:     Button btn;
9:     public void init( )
10:    {
11:        size = new Choice( );
12:        size.add("10");
13:        size.add("14");
14:        size.add("18");
15:        add(size);
16:        btn = new Button("효과");
17:        add(btn);
18:        size.addItemListener(this);
19:    }
20:    public void itemStateChanged(ItemEvent e)
21:    {
22:        Choice temp;
23:        Font oldF;
24:        String s;
25:        int si;
26:        if(e.getItemSelectable( ) instanceof Choice)
27:        {
28:            oldF = btn.getFont( );
29:            temp = (Choice)(e.getItemSelectable( ));
30:            s = temp.getSelectedItem( );
31:            si = Integer.parseInt(s);
32:            btn.setFont(new Font(oldF.getName( ), oldF.getStyle( ), si));
33:        }
34:    }
35:}
```

프로그람설명

실례 8-10의 프로그람은 ItemListener대면을 실현한 Applet를 정의하고있으며 그것은 Choice객체인 size와 단추 btn을 포함하고있다. 11행은 Choice객체를 창조한다. 12-14행은 10, 14, 18의 3개 문자렬항목을 내리떨구기목록객체 size에 추가한다. 18행은 내리떨구기목록객체 size를 ItemListener대면을 실현한 감시자 Applet에 등록한다. 20-34행에서는 선택사건의 메쏘드 itemStateChanged( )를 처리한다. 29행은 우선 이 사건원천을 Choice객체로 강제형변환한다. 30행에서는 사용자가 선택한 항목문자렬을 얻는다. 31행에서 이 문자렬을 옹근수로 변환하며 32행에서 btn의 서체를 size에서 사용자가 지정한 서체크기로 설정한다.

그림 8-11. 실례 8-10의 실행결과

그림 8-11은 실례 8-10의 실행결과이다.

8.6.5. 목록

1) 창조

목록(List) 역시 선택항목을 렬거하여 사용자가 선택하게 한다. 목록은 여러개중에서 여러개를 선택할수 있다. 즉 다중선택을 허용한다. 목록의 창조시 매 항목을 선택하여(목록의 매 항목을 Item이라고 한다.) 목록에 추가하여야 한다. 아래에 실례를 보여준다.

```
MyList = new List(4, true);
MyList.add("평양");
MyList.add("원산");
```

은 2개의 주소항목을 포함하는 목록을 창조한다. List객체구성자의 첫번째 파라메터는 목록의 높이를 나타내며 한번에 여러개의 항목을 현시할수 있다. 두번째 파라메터는 목록이 다중선택을 허용하는가를 나타낸다. 즉 동시에 여러개의 항목을 선택할수 있다.

2) 상용메쏘드

List객체의 getSelectedItem( )메쏘드는 사용자가 선택한 선택항목의 본문을 귀환시킨다. 검사칸과 다른점은 목록에서 반복선택과 다중선택을 할수 있다는것이다. 그러므로 List객체는 getSelectedItems( )메쏘드를 가지며 이 메쏘드는 String형의 배렬을 귀환시킨다. 배렬의 매개 원소는 사용자에 의해 선택되는 선택항목이며 모든 원소는 사용자에 의하여 선택되는 모든 항목을 포괄한다.

getSelectedIndex( )메쏘드는 선택된 항목번호를 귀환시킨다. 목록에 가입한 첫번째 항목번호는 0이며 두번째 항목번호는 1이다. 그러므로 getSelectedIndexs( )메쏘드는 선택된 항목번호들로 구성된 옹근수형배렬을 귀환시킨다.

select(int index)와 deselect(int index)메쏘드는 지정된 번호의 항목을 선택하거나 선택을 취소할수 있다. add(String item)메쏘드와 add(String item, int index)메쏘드는 각각 item으로 지정한 항목을 목록의 맨 마지막에 추가하거나 목록의 지정된 위치에 추가한다. remove(String item)메쏘드와 remove(int index)메쏘드는 표식으로 지정한 항목이나 번호위치로 지정한 항목을 목록으로부터 삭제한다. 이 두 메쏘드는 프로그람이 목록에 포함되여있는 선택항목을 동적으로 조종할수 있게 한다.

3) 사건응답

목록은 2개의 사건을 발생시킬수 있다. 즉 사용자가 목록의 어떤 항목을 찰칵하여 그것을 선택할 때 ItemEvent클라스의 선택사건을 발생시키며 사용자가 목록의 어떤 항목을 두번 찰칵할 때 ActionEvent클라스의 동작사건을 발생시킨다.

만일 프로그람이 이 2가지 사건에 응답하자면 목록을 각각 ItemEvent의 감시자 ItemListener와 ActionEvent의 감시자 ActionListener에 등록해야 한다.

```
MyList.addItemListener(this);
MyList.addActionListener(this);
```

그다음 감시자대면을 실현한 클라스에서 각각 선택사건에 응답하는 메쏘드와 동작사건에 응답하는 메쏘드를 정의할수 있다. 즉

```
public void itemStateChanged(ItemEvent e);
                                        //한번 찰칵하는 선택사건에 응답
public void actionPerfomed(ActionEvent e);
                                        //두번 찰칵 찰칵하는 동작사건에 응답
```

이렇게 목록에서 한번 찰칵하거나 두번 찰칵하는 동작을 할 때 체계는 자동적으로 우의 두 메쏘드를 호출하여 상응한 선택 또는 동작사건을 처리한다. 보통 itemStateChanged(ItemEvent e)메쏘드에서 e.getItemSelectable( )메쏘드를 호출하여 이 선택사건을 발생시키는 List객체의 인용을 얻을수 있다. List객체의 getSelectedIndex( )메쏘드나 getSelectedItem( )메쏘드를 리용하면 사용자가 목록의 어느 항목을 선택하였는가를 쉽게 알아낼수 있다. Checkbox의 메쏘드와 같이 e.getItemSelectable( )의 귀환값을 먼저 List객체로 강제형변환한 다음에야 List클라스의 메쏘드를 호출할수 있다. 실례로

```
String s=((List)e.getItemSelectable( )).getSelectedItem( );
```

한편 actionPerformed(ActionEvent e)메쏘드에서 e.getSource( )를 호출하면 이 동작사건을 발생시키는 List객체인용을 얻을수 있다. 그러나 이것 역시 강제형변환을 진행해야 한다.

```
((List)e.getSource( ));
```

e.getActionCommand( )를 호출하면 사건항목의 문자렬표식자를 얻을수 있으며 목록의 단일선택인 경우에는

((List)e.getSource( )).getSelectedItem( );

을 집행한 실행결과와 같다. 주의해야 할것은 목록의 두번 찰칵사건은 한번 찰칵사건을 두번 진행하여 만들수 없다는것이다. 사용자가 목록의 항목을 두번 찰칵할 때 우선 한번 찰칵하는 항목사건을 발생한 다음에 두번 찰칵하는 동작사건을 발생시킨다. 만일 두개의 사건을 등록하는 감시자를 정의하면 itemStateChanged( )메소드와 actionPerformed( )메소드는 각각 선후순서에 의해 호출될것이다.

실례 8-11

```
Example 8-11 TestList.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestList extends Applet implements ActionListener, ItemListener
6: {
7:      List MyList;
8:      Label result;
9:      public void init( )
10:     {
11:         result = new Label("선택항목을 두번 찰칵찰칵하였습니다");
12:         MyList = new List(4, true);
13:         MyList.add("백두산");
14:         MyList.add("금강산");
15:         MyList.add("묘향산");
16:         MyList.add("칠보산");
17:         MyList.add("구월산");
18:         MyList.add("한나산");
19:         add(MyList);
20:         add(result);
21:         MyList.addActionListener(this);
22:         MyList.addItemListener(this);
23:     }
24:     public void actionPerformed(ActionEvent e)
25:     {
26:         if(e.getSource( ) == MyList)
27:             result.setText("선택항목을 두번 찰칵 찰칵하였습니다"
```

```
                                        + e.getActionCommand( ));
28:    }
29:    public void itemStateChanged(ItemEvent e)
30:    {
31:        List temp;
32:        String sList[ ];
33:        String mgr = new String("");
34:        if(e.getItemSelectable( ) instanceof List)
35:        {
36:            temp = (List)(e.getItemSelectable( ));
37:            sList = temp.getSelectedItems( );
38:            for(int i = 0; i < sList.length; i++)
39:                mgr = mgr+sList[i] + " ";
40:            showStatus(mgr);
41:        }
42:    }
43:}
```

그림 8-12는 실례 8-11의 실행결과이다.

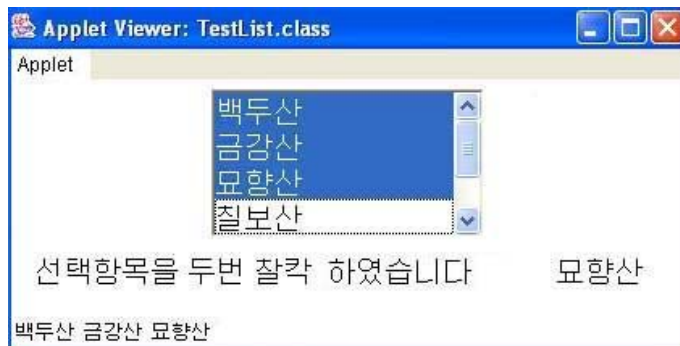



그림 8－12. **실례** 8－11 **의 실행결과**

프로그람설명

실례 8-11에서 정의한 Applet는 동작사건과 선택사건의 감시자로서 그것은 ActionListener와 ItemListener의 2개의 대면을 실현하고있다. 여기에 List객체 MyList와 자료를 현시하는데 쓰이는 표식자객체 result를 포함하고있다. 12-18행에서 한번에 4개의 항목을 현시하는 List객체 MyList에 6개의 문자렬항목(산이름)을 가입시켰다. 24-28행에서 정의한 actionPerform( )메쏘드는 사용자가 List항목을 두번 찰칵하는 동작사건에 응답하는데 쓰이며 사용자가 두번 찰칵한 항목문자렬을 표식자

result에 현시한다.

29-42행에서 정의한 itemStateChanged( )메쏘드는 사용자가 List항목을 한번 찰각하는 조작에 응답하는데 쓰이며 getSelectedItems( )메쏘드를 리용하여 이러한 항목들을 모두 한개의 문자렬로 조합하고 Applet의 상태띠에 현시한다.

40행의 메쏘드 showStatus( )는 Applet객체의 메쏘드로서 파라메터문자렬을 Applet의 상태띠에 현시한다.

목록의 다중선택경우의 처리는 비교적 복잡하다. 그러나 기본원리는 같다. 보통 목록사건에 응답하여야 할 처리상황이 많지 않으며 Checkbox에서처럼 프로그람이 관심하는것은 어떤 특정한 시각이다. 이것은 목록자체의 getSelectedIndex나 getSelectedItem메쏘드를 호출하여 얻을수 있다.

제7절. 흘림띠와 사건조종

- **Scrollbar는 련속적인 변화를 받아들이고 표현할수 있는 GUI부품이다.**
- **흘림띠의 사건관리자는 AdjustmentListener이다.**

8.7.1. 사건조종

AdjustmentEvent(사건조종)클라스는 ADJUSTMENT_VALUE_CHANGED사건만을 포함하고있다. ItemEvent사건이 일으키는 리상상태변화와 달리 ADJUSTMENT_VALUE_CHANGED는 GUI부품상태가 련속변화를 발생시키는 사건으로서 이 사건을 일으키는 구체적인 동작은 다음과 같다.

· 흘림띠(Scrollbar)를 조종하여 그의 위치를 변경시킨다.

· 사용자가 정의한 Scrollbar객체의 하위클라스부품을 조정하여 그의 위치를 변경시킨다.

AdjustmentEvent클라스의 주요메쏘드는 다음과 같다.

1) public Adjustable getAdjustable()

이 메쏘드는 상태변화사건을 일으키는 사건원천을 귀환시키며 상태변화사건을 일으킬수 있는 사건원천은 Adjustable대면을 실현한 클라스이다. 실례로 Scrollbar클라스는 Adjustable대면을 실현한 클라스이며 Adjustable대면은 java.awt패키지에서 정의한 대면이다. 메쏘드의 귀환값형은 대면이고 귀환시키는것은 대면을 실현한 클라스의 객체이다. 실례로 getAdjustable( )메쏘드를 호출하면 이 사건을 일으키는 Scrollbar객체의 인용을 귀환시킨다.

2) public int getAdjustmentType()

이 메쏘드는 상태변화사건의 상태변화형을 귀환시키며 그의 귀환값은

AdjustmentEvent클라스가 렬거하는 정적상수들중의 하나이다.

AdjustmentEvent.BLOCK_DECREMENT: 흘림칸을 찰칵하여 아래쪽으로 블로크상태감소를 일으키는 동작을 의미한다.

AdjustmentEvent.BLOCK_INCREMENT: 흘림칸을 찰칵하여 웃쪽으로 블로크상태증가를 일으키는 동작을 의미한다.

AdjustmentEvent.TRACK: 흘림띠의 미끄럼편을 시동하는 동작을 의미한다.

AdjustmentEvent.UNIT_PECREMENT: 흘림띠의 아래방향3각단추를 찰칵하여 최소단위감소를 일으키는 동작을 의미한다.

AdjustmentEvent.UNIT_INCREMENT: 흘림띠의 웃방향3각단추를 찰칵하여 최소단위증가를 일으키는 동작을 의미한다.

getAdjustmentType( )메쏘드의 호출과 그 귀환값을 통하여 사용자가 일으킨 어느 조작이 련속적으로 어느 상태변동을 일으키는가를 알수 있다.

3) public int getValue( )

getValue( )메쏘드는 상태변환후의 미끄럼편에 대응하는 수값을 귀환시킨다. 미끄럼편은 련속 조정할수 있는것이며 이 조정은 AdjustmentEvent사건을 일으킨다.

8.7.2. 흘림띠

1) 창조

흘림띠(Scrollbar)는 특수한 GUI부품으로서 그것은 련속적인 변화를 받아들이고 체현할수 있다. 이것을 《조정》이라고도 한다. Scrollbar클라스의 객체창조는 흘림칸, 증가화살표, 감소화살표, 흘림띠를 창조하는것으로 실현한다.

Scrollbar mySlider = new Scroolbar(Scroolbar.HORIZONTAL,50,1,0,100);

우의 명령문이 창조한 흘림띠는 그림 8-13과 같다.

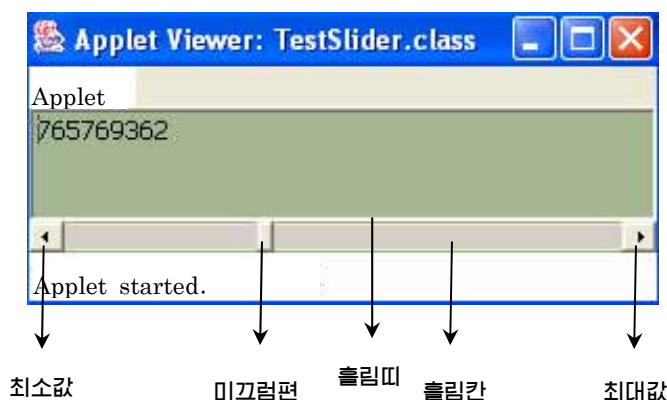



그림 8-13. 흘림띠의 매 구성부분

구성자의 첫번째 파라메터는 흘림띠의 류형으로서 Scrollbar.HORIZONTAL상수를 리용하면 수평흘림띠를 창조하며 Scrollbar.VERTICAL상수를 리용하면 수직흘림띠를 창조한다.

구성자의 두번째 파라메터는 미끄럼편의 처음 위치를 나타내는데 쓰이며 그것은 옹근수변수여야 한다. 구성자의 세번째 파라메터는 미끄럼편의 크기로서 미끄럼편크기와 전체 흘림칸길이의 비는 창문에서 볼수 있는 본문구역과 전체 본문구역과의 비와 같다. 미끄럼편에 대하여 본문구역이 흐르는 상태가 일어나지 않게 하려면 미끄럼편크기를 1로 놓을수 있다. 구성자의 네번째 파라메터는 흘림칸의 최소값이다. 구성자의 다섯번째 파라메터는 흘림칸의 최대값을 표현한다.

2) 상용메쏘드

새로 창조하는 흘림띠에 대하여 그것의 단위증분과 블로크증분을 설정할 때 아래의 메쏘드를 리용한다.

```
mySlider.setUnitIncrement(1);
mySlider.setBlockIncrement(50);
```

setUnitIncrement(int)메쏘드는 흘림띠의 단위증분을 지정한다. 이것은 사용자가 흘림띠 두 끝의 3각형단추를 찰칵할 때 나타내는 자료변경이다.

setBlockIncrement(int)메쏘드는 흘림띠의 블로크증분을 지정한다. 이것은 사용자가 흘림칸를 찰칵할 때 나타내는 자료변경이다.

흘림띠클라스는 우의 두 메쏘드와 상반되는 getUnitIncrement( )메쏘드와 getBlockIncrement( )메쏘드를 정의하여 흘림띠의 단위증분과 블로크증분을 각각 얻는다. getValue( )메쏘드는 미끄럼편위치를 나타내는 옹근수값을 귀환시키며 사용자가 흘림칸에서 미끄럼편의 위치를 변경시킬 때 getValue( )메쏘드의 귀환값도 대응한 값으로 변한다.

3) 사건응답

흘림띠는 AdjustmentEvent클라스가 나타내는 조정사건을 일으킬수 있으며 사용자가 각종 방식을 통하여 미끄럼편위치를 변경시킬 때 조정사건을 일으킬수 있다.

프로그람은 흘림띠가 일으키는 조정사건에 응답해야 하며 반드시 이 흘림띠를 AdjustmentListener대면을 실현한 조정사건감시자에 먼저 등록해야 한다.

```
mySlider.addAdjustmentListener(this);
```

조정사건감시자에서 조정사건에 응답하는 메쏘드는 다음과 같다.

```
public void adjustmentValueChanged(AdjustmentEvent e);
```

이 메쏘드는 보통 e.getAdjustable( )을 호출하여 현재 조정사건을 일으키는 사건원천을 얻는다. 다른 하나의 메쏘드는 AdjustmentEvent클라스의 getValue( )메쏘드로서 흘림띠의 getValue( )메쏘드와 기능이 같다. e.getValue( )메쏘드는 조정사건후의 미끄럼편위치를 나타내는 수값을 귀환시키며 e.getAdjustmentType( )메쏘드는 현재 조정사건의 류형을 귀한시킨다. 즉 사용자가 어느 방식을 사용하여 미끄럼편의 위치

를 변경시켰는가를 알수 있다. 이 메쏘드의 귀환값을 AdjustmentEvent클라스의 몇개 정적상수와 서로 비교하면 알수 있다.

AdjustmentEvent.BLOCK_INCREMENT: 블로크증가
AdjustmentEvent.BLOCK_DECREMENT: 블로크감소
AdjustmentEvent.UNIT_INCREMENT: 단위증가
AdjustmentEvent.UNIT_DECREMENT: 단위감소
AdjustmentEvent.TRACK: 마우스를 리용하여 미끄럼편이동을 시동

실례 8-12는 흘림띠를 창조하고 사용하는 례제이다.

실례 8-12

```
Example 8-12 TestSlider.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestSlider extends Applet implements AdjustmentListener
6: {
7:     Scrollbar mySlider;
8:     TextField sliderValue;
9:
10:    public void init( )
11:    {
12:        setLayout(new BorderLayout( ));
13:        mySlider = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, Integer.MAX_VALUE);
14:        mySlider.setUnitIncrement(1);
15:        mySlider.setBlockIncrement(50);
16:        add("South", mySlider);
17:        mySlider.addAdjustmentListener(this);
18:        sliderValue = new TextField(30);
19:        add("Center", sliderValue);
20:    }
21:    public void adjustmentValueChanged(AdjustmentEvent e)
22:    {
23:        int value;
24:        if(e.getAdjustable( ) == mySlider)
25:        {
26:            value = e.getValue( );
27:            sliderValue.setText(new Integer((int)value).toString( ));
```

```
28:            sliderValue.setBackground(new Color(value));
29:        }
30:    }
31:}
```

프로그람설명

실례 8-12의 프로그람은 AdjustmentListener대면을 실현한 Applet를 정의하고있으며 Scrollbar객체 mySlider와 본문마당객체 sliderValue를 포함하고있다. 12행은 Applet의 배치방안을 BorderLayout로 설정하며 13행은 mySlider객체를 창조한다. 14, 15행은 이 흘림띠의 련관속성을 설정한다. 16행은 그것을 Applet안에 추가한다. 17행은 그것을 조정사건의 감시자에게 등록한다. 21-31행이 정의한 adjustmentValueChanged( )메쏘드는 우선 조정사건을 일으키는 사건원천이 mySlider객체인가를 판단하고 다음에 getValue( )메쏘드를 호출하여 조정후 흘림띠가 지정하는 수값을 얻는다. 27행에서 본문마당객체에 의해 이 수값을 현시한다. 28행은 본문마당의 배경색을 흘림띠수값에 대응하는 색갈로 설정한다.

실례 8-12의 실행결과는 그림 8-13에서 보여주었다.

제8절. 화판과 마우스, 건반사건

- MouseEvent와 KeyEvent는 InputEvent클라스의 하위클라스이다.
- Canvas는 그림을 그리는데 쓰이는 4각형배경부품이다.

8.8.1. 마우스사건

MouseEvent클라스와 KeyEvent클라스는 모두 InputEvent클라스의 하위클라스이다. InputEvent클라스는 임의의 구체적인 사건은 포함하지 않으며 getModifiers( )메쏘드를 호출하여 귀환값을 자기의 몇개 정적옹근수형상수 ALT_MASK, CTRL_MASK, SHIFT_MASK, META_MASK, BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK와 비교하여 사용자가 기능건을 눌렀는가 혹은 마우스의 어느 건을 눌렀는가를 알아낼수 있다.

MouseEvent클라스는 정적상태옹근수형상수를 리용하여 몇가지 마우스사건을 나타낸다.

- MOUSE_CLICKED: 마우스찰칵사건을 나타낸다.
- MOUSE_DRAGGED: 마우스끌기사건을 나타낸다.
- MOUSE_ENTERED: 마우스진입사건을 나타낸다.

· MOUSE_EXITED: 마우스탈퇴사건을 나타낸다.
· MOUSE_MOVED: 마우스이동사건을 나타낸다.
· MOUSE_PRESSED: 마우스단추누르기사건을 나타낸다.
· MOUSE_RELEASED: 마우스단추해방사건을 나타낸다.

MouseEvent객체의 getID( )메쏘드귀환값을 우의 매 상수와 비교하여보면 사용자가 일으킨 사건이 어느 마우스사건인가를 구체적으로 알수 있다. 실례로 mouseEvt가 MouseEvent클라스의 객체라고 하면 아래의 명령문은 그것이 나타내는 사건이 MOUSE_CLICKED인가를 판단한다.

```
if(mouseEvt.getID( ) == MouseEvent.MOUSE_CLICKED)
```

그런데 일반적으로 이러한 처리를 진행할 필요는 없다. 그것은 MouseEvent사건감시자인 MouseListener와 MouseMotionListener에는 7개의 메쏘드가 있는데 각각 우의 7개 구체적인 마우스사건의 류형을 구별하고 련관있는 메쏘드를 자동호출하기때문에 이 메쏘드에 관련사건을 처리하는 코드만을 작성해놓을수 있다.

MouseEvent클라스에는 다음과 같은 중요한 메쏘드가 있다.

· public int getX( ): 마우스사건을 발생시키는 X자리표를 귀환한다.
· public int getY( ): 마우스사건을 발생시키는 Y자리표를 귀환한다.
· public Point getPoint( ): Point객체를 귀환, 마우스사건이 발생하는 자리표점을 귀환한다.
· public int getClickCount( ): 마우스찰칵사건의 찰칵회수를 귀환한다.

앞에서 언급한 MouseListener와 MouseMotionListener의 몇개의 구체적인 사건처리메쏘드는 MouseEvent클라스의 객체로서의 형식파라메터이다. 우에서 지적한 MouseEvent클라스의 메쏘드들을 호출하여 마우스사건을 일으키는 구체적인 정보를 얻을수 있다.

실례 8-13

```
Example 8-13 ResponseToMouse.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class ResponseToMouse extends Applet
6:                          implements MouseListener, MouseMotionListener
7: {
8:     public void init( )
9:     {
10:        this.addMouseListener(this);
```

```
11:         this.addMouseMotionListener(this);
12:     }
13:     public void mouseClicked(MouseEvent e)
14:     {
15:         if(e.getClickCount( ) == 1)
16:             showStatus("당신은 자리표 (" + e.getX( ) + ", " + e.getY( ) +
                                    ")에서 마우스를 찰칵하였습니다.");
17:         else if(e.getClickCount( ) == 2)
18:             showStatus("당신은 자리표 (" + e.getX( ) + ", " + e.getY( ) +
                                    ")에서 마우스를 두번 찰칵하였습니다.");
19:     }
20:     public void mouseEntered(MouseEvent e)
21:     {
22:         showStatus("마우스의 Applet에로의 진입.");
23:     }
24:     public void mouseExited(MouseEvent e)
25:     {
26:         showStatus("마우스의 Applet에서의 탈퇴.");
27:     }
28:     public void mousePressed(MouseEvent e)
29:     {
30:         showStatus("당신은 마우스를 눌렀습니다.");
31:     }
32:     public void mouseReleased(MouseEvent e)
33:     {
34:         showStatus("당신은 마우스를 해방시켰습니다.");
35:     }
36:     public void mouseMoved(MouseEvent e)
37:     {
38:         showStatus("마우스를 이동시켜 새로운 위치 (" + e.getX( ) + ", "
                                        + e.getY( ) + ")에 있습니다.");
39:     }
40:     public void mouseDragged(MouseEvent e)
41:     {
42:         showStatus("마우스를 끌기하였습니다.");
43:     }
44:}
```

프로그람설명

실례 8-13에서는 MouseListener와 MouseMotionListener대면을 실현한 Applet를 정의하고있으며 10, 11행은 이 Applet를 각각 마우스사건과 마우스동작사건의 감시자에게 등록한다. 13-35행에서 정의한 5개의 메쏘드는 MouseListener대면에 대하여 정의한 5개의 같은 이름을 가진 추상메쏘드의 구체적인 실현이다. 36-43행에서 정의한 2개의 메쏘드는 MouseMotionListener대면에서 정의한 2개의 같은 이름을 가진 추상메쏘드에 대한 구체적인 실현이다. 프로그람에서는 모든 마우스사건을 감시하다가 해당한 사건정보를 Applet의 상태띠에 현시한다.

8.8.2. 건반사건

KeyEvent클라스는 아래와 같은 3개의 구체적인 건반사건을 가진다. 이것들은 KeyEvent클라스의 몇개 같은 이름들을 가진 정적옹근수형상수에 각각 대응한다.

- KEY_PRESSED: 건반을 누르고있는 사건을 의미한다.
- KEY_RELEASED: 건반해방사건을 의미한다.
- KEY_TYPED: 건반을 눌렀다는 사건을 의미한다.

KeyEvent클라스의 주요메쏘드인 public char getKeyChar( )는 건반사건을 일으킨 건반에 대응하는 Unicode문자를 귀환시킨다. 만일 Unicode문자에 대응하지 않으면 KeyEvent클라스의 정적상수 KeyEvent.CHAR_UNDEFINED를 귀환시킨다.

KeyEvent사건에 대응하는 감시자대면은 KeyListener이다. 이 대면은 아래의 3개 추상메쏘드를 정의하고있으며 KeyEvent의 3개의 사건류형에 대응한다.

- public void keyPressed(KeyEvent e);
- public void keyReleased(KeyEvent e);
- public void keyTyped(KeyEvent e);

보는바와 같이 사건클라스의 사건류형이름은 대응하는 감시자대면의 추상메쏘드이름과 아주 류사하며 또한 량자간의 응답관계를 표현하고있다. KeyListener대면의 클라스를 실현하기만 하면 반드시 우의 3개의 추상메쏘드를 구체적으로 실현할수 있다. 3개의 구체적인 사건에 대한 사용자프로그람의 응답코드를 메쏘드본체에 작성하며 이 코드안에서 실제파라메터 KeyEvent객체 e의 몇가지 정보를 사용한다. 이것은 e의 메쏘드(례: getSource( ), getKeyChar( ) 등)를 호출하여 실현하여야 한다. 실례로 아래의 명령문은 사용자가 y건을 입력하였는가 n건을 입력하였는가를 판단하여 긍정 혹은 부정의 대답을 나타낸다.

```
public void keyPressed(KeyEvent e)
{
    char ch = e.getKeyChar( );
    if(ch == 'y' || ch == 'Y')
        m_result.setText("긍정");
    else if(ch == 'n' || ch == 'N')
```

```
                m_result.setText("부정");
            else
                m_result.setText("입력할 메쏘드가 없다");
        }
```

여기서 m_result는 정보를 출력하는데 쓰이는 Label객체이다.

8.8.3. 화판

화판(Canvas)은 그림을 그리는데 쓰이는 4각형배경부품으로서 Applet안에서 각종 그림을 그려낼수 있고 마우스와 건반사건에 응답할수 있다.

1) 창조

Canvas의 구성자는 파라메터를 가지지 않는다.

```
Canvas myCanvas = new Canvas( );
```

Canvas객체를 창조한 다음에는 반드시 setSize( )메쏘드를 호출하여 이 화판객체의 크기를 확정하여야 하며 그렇지 않으면 사용하는 실행대면에서 이 화판을 볼수 없다.

2) 상용메쏘드

Canvas는 public void paint(Graphics g)메쏘드만을 가지며 사용자프로그람이 이 메쏘드를 재정의하면 Canvas에서 련관도형들을 그려낼수 있다.

3) 사건생성

Canvas객체는 Applet와 류사하게 건반과 마우스사건을 일으킬수 있다.

실례 8-14

```
Example 8-14 TestCanvas.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestCanvas extends Applet
6: {
7:      CanvasDraw cd;
8:
9:      public void init( )
10:     {
11:         cd = new CanvasDraw(new Dimension(200, 180), this);
12:         cd.setBackground(Color.pink);
13:         add(cd);
14:         cd.requestFocus( );
```

```
15:     }
16:}
17:class CanvasDraw extends Canvas implements KeyListener
18:{
19:     Applet m_Parent;
20:     boolean m_bFlag = false;
21:     int currentX = 0, currentY = 0, startX = 0, startY = 0;
22:     StringBuffer sb = new StringBuffer( );
23:     CanvasDraw(Dimension d, Applet p)
24:     {
25:          m_Parent = p;
26:          setSize(d);
27:          setBackground(Color.gray);
28:          addKeyListener(this);
29:          addMouseListener(new MouseAdpt(this));
30:          addMouseMotionListener(new MouseMotionAdpt(this));
31:     }
32:     void setStart(int x, int y)
33:     {
34:          startX = x;
35:          startY = y;
36:     }
37:     void setCurrent(int x, int y)
38:     {
39:          currentX = x;
40:          currentY = y;
41:     }
42:     void setMouseDragged(boolean b)
43:     {
44:          m_bFlag = b;
45:     }
46:     void showMeg(String s)
47:     {
48:          m_Parent.showStatus(s);
49:     }
50:     void clearAll( )
51:     {
52:          startX = 0;
53:          startY = 0;
```

```
54:         currentX = 0;
55:         currentY = 0;
56:         repaint( );
57:     }
58:     public void keyTyped(KeyEvent e)
59:     {
60:         char ch = e.getKeyChar( );
61:         sb.append(ch);
62:         showMeg("건 " + sb.toString( ) + "을 누르기");
63:         repaint( );
64:     }
65:     public void keyPressed(KeyEvent e){ }
66:     public void keyReleased(KeyEvent e){ }
67:     public void paint(Graphics g)
68:     {
69:         g.drawString("(" + currentX + ", " + currentY + ")" + sb, 10, 20);
70:         if(m_bFlag)
71:             g.drawLine(startX, startY, currentX, currentY);
72:     }
73:}
74:class MouseAdpt extends MouseAdapter
75:{
76:     CanvasDraw m_Parent;
77:     MouseAdpt(CanvasDraw p)
78:     {
79:         m_Parent = p;
80:     }
81:     public void mousePressed(MouseEvent e)
82:     {
83:         m_Parent.setStart(e.getX( ), e.getY( ));
84:         m_Parent.showMeg("선그리기 시작");
85:     }
86:     public void mouseReleased(MouseEvent e)
87:     {
88:         m_Parent.showMeg("직선을 그렸습니다.");
89:     }
90:     public void mouseEntered(MouseEvent e)
91:     {
92:         m_Parent.showMeg("마우스의 화판에로의 진입");
```

```
93:     }
94:     public void mouseExited(MouseEvent e)
95:     {
96:         m_Parent.showMeg("마우스의 화판에서의 탈퇴");
97:     }
98: }
99: class MouseMotionAdpt extends MouseMotionAdapter
100:{
101:     CanvasDraw m_Parent;
102:     MouseMotionAdpt(CanvasDraw p)
103:     {
104:         m_Parent = p;
105:     }
106:     public void mouseMoved(MouseEvent e)
107:     {
108:         m_Parent.setCurrent(e.getX( ), e.getY( ));
109:         m_Parent.setMouseDragged(false);
110:         m_Parent.repaint(10, 0, 60, 30);
111:     }
112:     public void mouseDragged(MouseEvent e)
113:     {
114:         m_Parent.setCurrent(e.getX( ), e.getY( ));
115:         m_Parent.setMouseDragged(true);
116:         m_Parent.repaint( );
117:     }
118:}
```

그림 8-14는 실례 8-14의 실행결과이다.

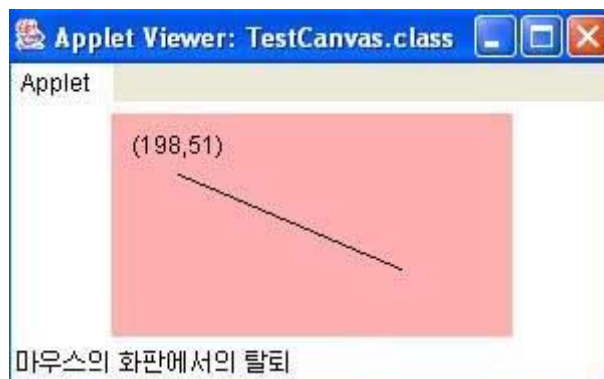



그림 8-14. 실례 8-14의 실행결과

프로그람설명

이 프로그람은 모두 4개의 클라스를 정의하고있다. 여기서 CanvasDraw는 사용자가 정의한 체계클라스 Canvas의 하위클라스이며 19행에서 정의한 Applet객체m_Parent는 CanvasDraw객체를 포함하는 Applet객체이다. 또한 론리값형변수 m_bFlag는 사용자가 마우스를 리용하여 직선을 그리는 상태에 대한 표식이며 startX와 startY는 직선을 그리는 시작점의 자리표, currentX와 currentY는 현재의 마우스위치이다. String Buffer객체 sb는 사용자가 건입력한 문자를 구분하는데 쓰인다. 주의할것은 Canvas객체와 그의 하위클라스객체는 기본부품이지 용기가 아니라는것이다.

CanvasDarw객체는 마우스사건과 건반사건에 응답할수 있으며 KeyListener대면을 실현한다. 여기서는 3개의 추상메쏘드를 재정의하고있는데 KeyTyped( )메쏘드에 대해서만 본체를 작성한다. 이제 사용자가 건입력한 문자는 sb객체에 보존되며 그 자료가 상태띠에 현시된다. MouseListener와 MouseMotionListener에서의 추상메쏘드는 비교적 많다. 만일 CanvasDraw가 이 두개의 대면을 실현하였으면 반드시 7개의 추상메쏘드를 재정의해야 하지만 프로그람에서는 실제상 2~3개의 메쏘드만을 사용하였다. 이 문제를 해결하기 위하여 실례 8-14에서는 사건오림클라스 MouseAdapter와 MouseMotionAdapter를 사용하였다.

매개 감시자대면은 사건오림클라스에 대응하며 사건오림은 빈 메쏘드본체를 리용하여 감시자대면에 대응하는 메쏘드를 구체적으로 실현하고있다. MouseAdapter는 아래와 같이 정의된다.

```
public class MouseAdapter implements MouseListener
{
    public void mouseClicked(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}
```

이렇게 MouseAdapter클라스가 상위클라스여야 사용하려는 메쏘드만을 재정의할수 있고 프로그람이 간단하게 된다. 실례 8-14에서 정의한 MouseAdpt클라스와 MouseMotionAdpt클라스는 바로 전문적으로 마우스사건응답에 쓰이는 오림클라스의 하위클라스이다. 프로그람은 마우스의 현위치를 현시하고 마우스의 시동조작에 응답하며 Canvas에서 직선을 그린다.

TestCanvas클라스는 프로그람의 주클라스로서 여기에는 한개의 CanvasDraw의 객체가 포함된다.

제9절. 배치설계

- 배치편성클라스: LayoutManager, FlowLayout, BorderLayout, CardLayout, GridLayout, GridBagLayout, NullLayout
- 메쏘드: setLayout(LayoutManager mgr);

앞의 례제에서는 기본부품이 하나의 용기에 간단히 추가되였다. 만일 용기에서 부품의 위치를 어떻게 설정하고 조절하겠는가를 더 고려하려면 배치설계에 대한 지식을 학습하여야 한다.

Java의 GUI대면부설계에서 배치조종은 용기에 대한 배치편성기를 설치하여 실현한다. Java.awt패키지에서는 모두 5개의 배치편성클라스를 정의하고있다. 매개 배치편성클라스에는 하나의 배치편성기가 대응되는데 그것에는 FlowLayout, BorderLayout, CardLayout, GridLayout와 GridBagLayout가 있다. 하나의 용기가 하나의 배치방안을 설정할 때 그것은 방안에 대응하는 배치편성클라스의 객체를 창조하여 이 객체를 자기의 배치편성기로 설정하여야 한다. 용기에서 배치편성기를 설정하지 않으면 여기의 객체는 가리우기될수 있으므로 사용에 영향을 미치게 된다. 그러므로 반드시 매개 용기에 대하여 적합한 배치편성기를 설치하여야 한다. 아래에서 몇가지 배치편성기를 상세히 서술한다.

8.9.1. FlowLayout

FlowLayout는 용기 Panel과 그것의 하위클라스 Applet가 기정으로 사용하는 배치편성기이다. 만일 Panel이나 Applet에 대하여 배치편성기를 지정하지 않으면 기정으로 FlowLayout에 대응하는 배치방안을 사용한다.

FlowLayout에 대응하는 배치방안은 아주 간단하다. 이 방안을 리용한 용기는 여기에 부품을 추가하는 선후순서에 따라 왼쪽에서 오른쪽으로 배렬하며 한행을 완전히 배렬한후에 다음행으로 바꾸어 왼쪽에서 오른쪽으로 계속 배렬한다. 매 행의 부품은 배렬속에 있게 된다. 부품이 많지 않을 때 이 방안의 사용은 매우 편리하다.

FlowLayout를 사용하는 용기에 부품을 추가하려면 아래의 명령을 사용한다.

```
add(부품이름);
```

부품은 용기에서 순차로 배렬되며 FlowLayout의 배치능력이 제한되여있으므로 이 경우에 용기겹치기될수 있다.

FlowLayout배치편성기를 사용하지 않은 원본용기를 FlowLayout로 바꾸려면 아래의 명령문을 사용한다.

```
setLayout(new FlowLayout( ));
```

setLayout( )메쏘드는 용기들의 상위클라스 Container의 메쏘드이며 용기에 대한 배치편성기를 설정하는데 쓰인다. FlowLayout클라스의 객체창조는 우의 명령문에서처럼 파라메터가 없는 구성자를 사용할수도 있고 아래의 두 구성자를 사용할수도 있다.

1) **FlowLayout (int align, int hgap, int vgap)**

파라메터 align은 매행 부품의 자리맞추기방식을 지정하는데 3개의 정적상수 LEFT, CENTER, RIGHT중의 하나를 취할수 있다.

파라메터 hgap와 vgap는 각각 매 부품사이의 가로세로간격을 화소단위로 지정한다.

2) **FlowLayout(int align)**

파라메터 align은 매행 부품의 자리맞추기방법을 지정하며 부품사이의 가로세로간격은 5pixel로 고정한다.

파라메터가 없는 구성자가 창조하는 FlowLayout객체는 자리맞추기방식이 CENTER상수로 지정한 중심맞추기이며 부품사이의 가로세로간격은 5pixel이다.

8.9.2. BorderLayout

BorderLayout 역시 일종의 간단한 배치편성기이다. 이 배치방안에서는 용기안의 공간을 간단히 동, 서, 남, 북, 중심의 5개 구역(그림 8-15)으로 가르고 매개에 하나의 부품을 추가한다. 이때 부품을 어느 구역에 추가하겠는가를 지적해야 한다.

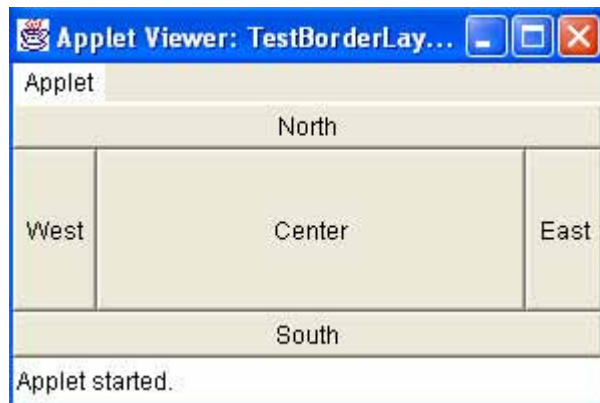



그림 8-15. 실례 8-15의 실행결과

북부와 남부구역에 분포되는 부품은 가로방향으로 용기의 전체 길이까지 확장하며 동부와 서부에 분포되는 부품은 용기의 나머지부분의 너비까지 늘이고 마지막 남은 부품은 중심구역에 배치한다. 만일 어떤 구역에 부품을 배치하지 않으면 다른 부품이 그 공간을 차지할수 있다. 실례로 북부에 부품을 배치하지 않았으면 서부와 동부의 부품이 웃쪽으로 용기의 제일 웃쪽까지 확장되며 서부와 동부에 부품이 배치되지 않았으면 중심에 위치한 부품이 용기의 좌우변두리까지 확장될수 있다.

파라메터가 없는 구성자를 사용하여 창조한 BorderLayout는 매 부품사이의 가로세로간격이 령이다는것을 의미한다. 매 부품사이에 간격을 두려면 다음의 BorderLayout구성자를 사용할수 있다.

BorderLayout(int hgap, int vgap);

BorderLayout는 오직 5개의 구역만을 지정할수 있다. 만일 용기에 5개이상의 부품을 배치하려면 반드시 용기의 겹치기를 사용하거나 다른 배치방안을 리용하여야 한다. 실례 8-15에서는 BorderLayout를 사용하여 5개의 부품들을 배치하였다.

실례 8-15

```
Example 8-15 TestBorderLayout.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestBorderLayout extends Applet
6: {
7:      Button north, south, west, east, center;
8:
9:      public void init( )
10:     {
11:         setLayout(new BorderLayout( )); // BorderLayout배치편성기설치
12:         north = new Button("North");
13:         south = new Button("South");
14:         west = new Button("West");
15:         east = new Button("East");
16:         center = new Button("Center");
17:         add("East", east);     //부품추가시 구역분배
18:         add("Center", center);
19:         add("North", north);
20:         add("South", south);
21:         add("West", west);
22:     }
23:}
```

프로그람설명

실례 8-15의 실행결과는 그림 8-15에서 보여주었다. 이 실례의 Applet객체에서는 BorderLayout배치편성기를 리용하여 5개의 위치에 각각 5개단추를 추가하였다.

8.9.3. CardLayout

CardLayout를 사용하는 용기들은 표면상 여러개의 부품들을 겹치게 할수 있다. 그러나 실제상 어떤 시각에 용기는 이 부품들중 한개만을 선택하여 현시하며(례: 《주패놀이》에서 제일웃쪽에 한장만을 현시할수 있는것과 같이) 이 현시된 부품은 모든 용기공간을 차지하게 된다. CardLayout를 사용하는 일반적인 단계는 아래와 같다.

배치편성기인 CardLayout객체를 창조한다.

Mycard = new CardLayout( );

용기의 setLayout( )메쏘드를 사용하여 용기에 배치편성기를 설치한다.

setLayout(Mycard);

용기의 add( )메쏘드를 호출하여 부품을 용기에 추가하며 동시에 부품에 문자렬이름을 준다. 앞으로 배치편성기는 이 이름에 기초하여 부품을 호출하고 현시한다.

add(문자렬, 부품);

CardLayout의 show( )메쏘드를 호출하여 문자렬이름(show(용기이름, 문자렬))에 따라 부품을 현시하거나 부품을 용기에 추가시키는 순서에 따라 부품을 현시한다. 즉 first(용기이름)메쏘드는 첫번째 부품을 현시하며 last(용기이름)메쏘드는 제일 마지막 부품을 현시한다.

실례 8-16은 CardLayout의 배치방안사용례이며 그림 8-16은 그 실행결과이다.

실례 8-16

```
Example 8-16 TestCardLayout.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestCardLayout extends Applet
6: {
7:       CardLayout MyCard = new CardLayout( ); //CardLayout배치편성기객체창조
8:       Button btn1 = new Button("1페지");
9:       Button btn2 = new Button("2페지");
10:      Button btn3 = new Button("3페지");
11:      Button btn4 = new Button("4페지");
12:      Button btn5 = new Button("5페지");
```

```
13:
14:    public void init( )
15:    {
16:        setLayout(MyCard);  //용기의 배치방안을 CardLayout로 설정
17:        add("1페지", btn1);   //부품추가 및 이름지정
18:        add("2페지", btn2);
19:        add("3페지", btn3);
20:        add("4페지", btn4);
21:        add("5페지", btn5);
22:        btn1.addMouseListener(new MouseMoveCard(MyCard, this));
23:        btn2.addMouseListener(new MouseMoveCard(MyCard, this));
24:        btn3.addMouseListener(new MouseMoveCard(MyCard, this));
25:        btn4.addMouseListener(new MouseMoveCard(MyCard, this));
26:        btn5.addMouseListener(new MouseMoveCard(MyCard, this));
27:    }
28:}
29:class MouseMoveCard extends MouseAdapter
30:{
31:    CardLayout cl;
32:    Applet m_Parent;
33:    MouseMoveCard(CardLayout c, Applet a)
34:    {
35:        cl = c;
36:        m_Parent = a;
37:    }
38:    public void mouseClicked(MouseEvent e)
39:    {
40:        if(e.getModifiers( ) == InputEvent.BUTTON1_MASK)
41:            cl.next(m_Parent);
42:        else
43:            cl.previous(m_Parent);
44:    }
45:}
```

프로그람설명

실례 8-16에서 정의하고있는 Applet는 5개의 단추부품을 가지고있다. 16행은 7행에서 창조한 CardLayout객체 MyCard를 Applet의 배치방안으로 설정한다. 5개의

단추는 마우스사건에 응답하며 왼쪽건을 찰칵하면 다음 단추가 튀여나오고 오른쪽건을 찰칵하면 앞의 단추가 튀여나온다.

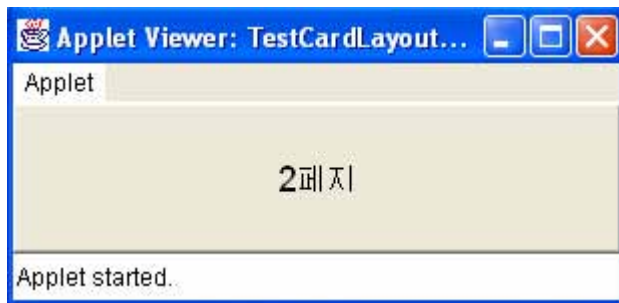



그림 8-16. 실례 8-16의 실행결과

8.9.4. GridLayout

GridLayout는 비교적 많이 사용하는 배치편성기이다. 그의 기본적인 배치방안은 용기의 공간을 행과 렬의 개수를 곱한만큼의 그물구역으로 가르는것이다. 부품은 바로 이렇게 분할해놓은 작은 그물구역들에 위치하게 된다.

GridLayout는 비교적 능동적이고 프로그람에서 그물들을 자유롭게 조종할수 있을뿐아니라 부품의 위치 역시 비교적 정확하다. GridLayout배치편성기를 사용하는 일반적인 단계는 아래와 같다.

배치편성기인 GridLayout객체를 창조한다. 그물을 나누는 행수와 렬수를 지정하고 용기의 setLayout()메쏘드를 사용하여 용기에 이 배치편성기를 설치한다:

setLayout(new GridLayout(행수, 렬수))

용기의 add()메쏘드를 호출하여 부품을 용기에 추가한다.

용기에서 부품의 배치는 1행의 첫칸으로부터 마지막칸으로, 2행의 첫칸으로부터 마지막칸으로, …, 마지막행의 마지막칸의 순서로 진행한다. 매 그물에 반드시 부품을 추가하여야 하며 만일 어떤 그물을 비게 하려면 그것에 빈 표식자를 추가시켜야 한다. 즉 add(new Label())로 표시한다.

실례 8-17

```
Example 8-17 TestGridLayout.java
1: import java.applet.*;
2: import java.awt.*;
3: import java.awt.event.*;
4:
5: public class TestGridLayout extends Applet
6: {
7:      public void init()
8:      {
```

```
9:          setLayout(new GridLayout(5, 6)); // 그물의 5행, 6렬에 배치한다.
10:           for(int i = 0; i < 5; i++)
11:               for(int j = 0; j < 6; j++)
12:           {
13:               if((int)(Math.random( ) * 100) >= 50)
14:                   add(new Button(Integer.toString(i * 6 + j))); // 단추의 추가
15:               else
16:                   add(new Label( ));
17:           }
18:     }
19:}
```

실례 8-17에서는 GridLayout를 사용하고있으며 그림 8-17은 그 실행결과이다. 그림 8-17(ㄴ)는 우연수를 리용하여 단추부품을 우연적으로 생성한것이다. 그림 8-17(ㄱ)는 모든 구역을 단추로 채우는 경우이며 실례 8-17의 명령문

if((int)(Math.random( ) * 100) >= 50)을

if((int)(Math.random( ) * 100) >= 0)

으로 고쳐 얻을수 있다. 이로부터 GridLayout의 배치방안과 그물을 정확히 보고 구별할수 있다.

(ㄱ) (ㄴ)

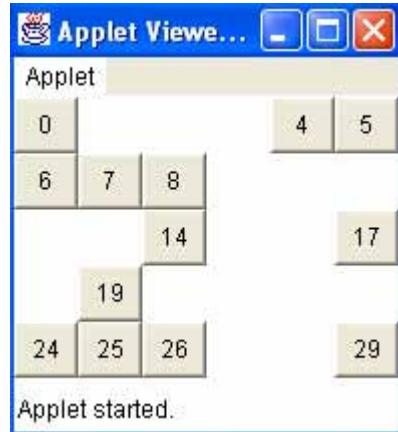



그림 8-17. 실례 8-17의 실행결과

8.9.5. NullLayout

앞의 배치방안들은 일정한 규칙으로 부품들을 배치한다. 프로그람작성자가 부품의 배치를 임의로 섬세하게 하자면 NullLayout를 사용하여야 한다.

setLayout(null)로 하면 된다. 그러나 부품을 배치하자면 추가하려는 부품의 자리표, 너비와 높이를 계산하여야 한다. 그리고 부품들사이의 위치도 고려하면서 부품들이 중복되지 않도록 해야 한다. 메쏘드는 다음과 같다.

setBounds(int x,int y,int width,int height)

실례 8-18

```
Example 8-18 NullLayout.java
1: import java.awt.*;
2: public class NullLayout extends Frame
3: {
4:      private Button b1, b2;
5:      private TextField t;
6:      private TextArea a;
7:      public NullLayout( )
8:      {
9:          super("NullLayoutTest");
10:         b1 = new Button("button1");
11:         b2 = new Button("button2");
12:         t = new TextField( );
13:         a = new TextArea( );
14:         load( );
15:     }
16:     public void load( )
17:     { //배치방안을 리용하지 않음
18:         setLayout(null);
19:         add(b1);add(b2);
20:         add(t);add(a);
21:         b1.setBounds(10, 30, 50, 20);
22:         b2.setBounds(10, 60, 50, 20);
23:         t.setBounds(10, 90, 30, 20);
24:         a.setBounds(10, 120, 200, 50);
25:         setSize(400, 200);
26:         setVisible(true);
27:     }
28:     public static void main(String args[])
29:     {
30:         NullLayout n = new NullLayout( );
31:     }
32:}
```

프로그람설명

실례에서는 배치방안을 리용하지 않고 용기에 두개의 Button, 한개의 TextField, 한개의 TextArea를 추가하였으며 setBounds( )메쏘드를 리용하였다.

그림 8-18은 실례 8-18의 실행결과이다.

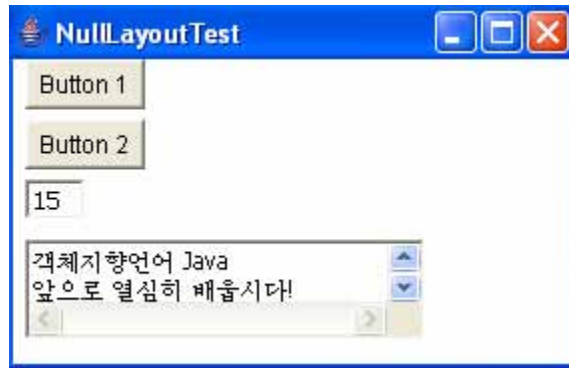



그림 8-18. 8-18의 실행결과

제10절. Panel과 용기사건

- Containner클라스의 메쏘드
  add( ), getComponent(int index)와 getComponent(int x, int y), remove(Component)와 remove(int index), removeAll( ), setLayout( )
- ContainerEvent클라스의 주요메쏘드
  getContainer( ), getChild( )
- Panel은 테두리가 없는 용기이며 단독적인 사용이 불가능하다.
- Panel은 다른 Panel을 또 포함할수 있다.

8.10.1. Container클라스

Container클라스는 추상클라스로서 모든 용기부품이 반드시 가지고있어야 할 메쏘드와 기능을 포함하고있다.

· add( ): Container클라스에는 재정의를 통한 여러개의 add( )메쏘드가 있으며 그의 작용은 모두 Component부품(기본부품일수도 있고 다른 용기부품일수도 있다.)을 현재의 용기에 추가하는것이다. 매개 용기에 추가된 부품은 추가한 선후차에 따라 번호를 가진다.

· **getComponent(int index)와 getComponent(int x, int y)**: 이 두 메쏘드는 각각 지정한 번호나 지정된 (x,y)자리표점을 가진 부품을 얻는데 리용한다.

· **remove(Component)와 remove(int index)**: 지정한 부품이나 지정된 번호를 가진 부품을 용기로부터 제거한다.

· **removeAll( )**: 용기의 모든 부품들을 제거한다.

· **setLayout( )**: 용기에 배치편성기를 설치한다.

Container는 ContainerEvent클라스가 나타내는 용기사건을 일으킬수 있다. 용기에 부품을 추가하거나 제거할 때 용기는 두 용기사건(COMPONENT_ADDED와 COMPONENT_REMOVED)을 각각 일으킨다. 용기사건에 응답하려면 프로그람에서 용기사건의 감시자대면 ContainerListener를 실현하여야 하며 감시자내부에서 대면의 용기사건을 처리하는데 쓰이는 2가지 메쏘드를 구체적으로 실현하여야 한다.

```
public void componentAdded(ContainerEvent e);
                    //용기에 부품을 추가하는 사건에 응답하는 메쏘드
public void componentRemoved(ContainerEvent e);
                    //용기로부터 부품을 제거하는 사건에 응답하는 메쏘드
```

이 두 메쏘드안에서 실제파라메터 e의 메쏘드 e.getContainer를 호출하여 사건을 일으키는 용기객체의 인용을 얻을수 있다. 이 메쏘드의 귀환형은 Container이다. 또한 e.getChild메쏘드를 호출하여 사건발생시 용기에 추가되거나 제거되는 부품을 얻을수 있으며 이 메쏘드의 귀환형은 Compenent이다.

8.10.2. 용기사건

ContainerEvent클라스는 용기와 련관된 2개의 구체적인 사건들을 포함하고있다.

· COMPONENT_ADDED: 부품을 현재의 용기객체에 추가한다.

· COMPONENT_REMOVED: 부품을 현재의 용기객체에서 제거한다.

ContainerEvent클라스의 주요메쏘드는 다음과 같다.

· public Container getContainer( ): 사건을 일으키는 용기객체를 귀환한다.

· public Component getChild( ): 사건을 일으킬 때 추가되거나 제거되는 부품객체를 귀환한다.

8.10.3. Panel

Panel은 테두리가 없는 용기에 속한다. 테두리가 없는 용기에는 Panel과 Applet가 있으며 여기서 Panel은 Container의 하위클라스이고 Applet는 Panel의 하위클라스이다.

Panel은 가장 간단한 용기로서 테두리나 볼수 있는 경계를 가지지 않으며 이동, 확대, 축소, 닫기를 할수 없다. 프로그람에서 Panel을 제일 바깥층의 용기로 사용할수 없으며 따라서 Panel은 다른 용기에 가입시켜야만 리용할수 있다.(례: Frame,

Applet 등) Panel역시 다른 Panel을 또 포함할수 있으며 Panel상에서는 용기가 겹칠 수 있다.

Panel을 사용하는 목적은 보통 도형사용자대면부의 매개 부품을 계승하고 관리하기 위해서이다. 따라서 용기에서 부품의 배치조작이 보다 편리하게 된다. 프로그람은 Panel의 크기를 지정할수 없으며 Panel크기는 여기에 포함된 모든 부품 그리고 그것을 포함하는 용기의 배치방안과 그 용기의 다른 부품들에 의하여 결정된다.

실례 8-19

```
Example 8-19 TestPanel.java
1: import java.awt.*;
2: import java.applet.*;
3: import java.awt.event.*;
4:
5: public class TestPanel extends Applet implements ActionListener, ContainerListener
6: {
7:     Panel p1, p2, p3;
8:     Label prompt1, prompt2, prompt3;
9:     Button btn;
10:    public void init( )
11:    {
12:        p1 = new Panel( );
13:        p1.setBackground(Color.gray);
14:        p2 = new Panel( );
15:        p2.setBackground(Color.red);
16:        p3 = new Panel( );
17:        p3.setBackground(Color.cyan);
18:        prompt1 = new Label("나는 첫번째 Panel안에 있습니다.");
19:        prompt2 = new Label("나는 두번째 Panel안에 있습니다.");
20:        prompt3 = new Label("나는 세번째 Panel안에 있습니다.");
21:        btn = new Button("in Panel3");
22:        p1.add(prompt1);
23:        p2.add(prompt2);
24:        p3.add(prompt3);
25:        p3.add(btn);
26:        p1.add(p3);
27:        add(p1);
28:        add(p2);
```

```
29:        btn.addActionListener(this);
30:        p1.addContainerListener(this);
31:    }
32:    public void actionPerformed(ActionEvent e)
33:    {
34:        if(e.getSource( ) == btn)
35:            p1.remove(p3);
36:    }
37:    public void componentRemoved(ContainerEvent e)
38:    {
39:        showStatus("세번째 Panel을 제거하였습니다.");
40:    }
41:    public void componentAdded(ContainerEvent e){ }
42:}
```

그림 8-19는 실례 8-19의 실행결과이다.

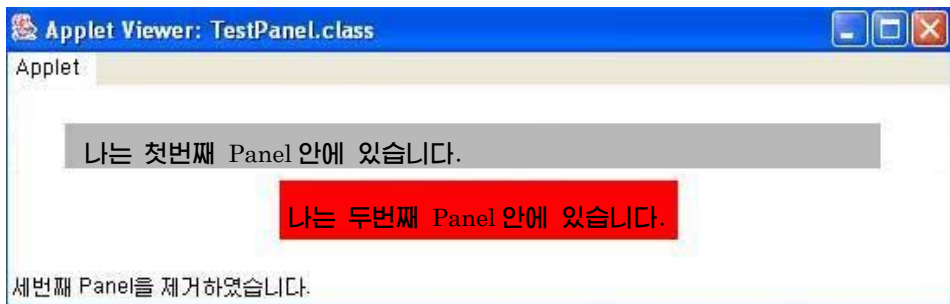
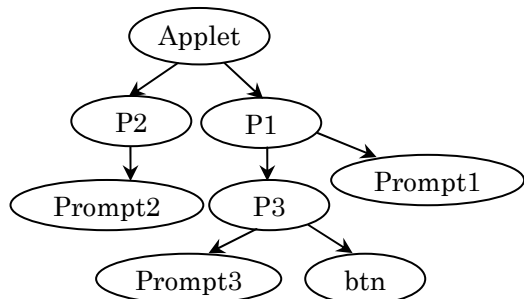



그림 8-19. 실례 8-19의 실행결과

Java프로그람에서 GUI대면부를 실현할 때 먼저 매 용기들사이의 포함 겹치기관계를 명백히 해야 한다. 실례 8-19에서의 GUI에는 그림 8-20과 같은 포함계층관계가 존재한다.

그림 8-20. 여러용기도형대면의 구조계층관계

이 프로그람은 Java Applet이므로 프로그람의 제일 바깥층의 용기는 Applet이며 여기에 두개의 부품을 포함하고있다. 즉 첫째 부품은 panel객체 p1이고 둘째 부품은 다른 panel객체인 p2이다. p1에는 표식자 prompt1과 세번째 panel객체 p3이 포함되여있고 p3에는 표

식자 prompt3과 단추부품 btn이 포함되여있다. 여기서 모든 용기는 기정인 FlowLayout배치방안을 사용하고있다.

사용자가 단추부품 btn을 찰칵할 때 프로그람은 p1로부터 p3을 제거하고 CONTAINER_REMOVED사건을 일으킨다. 프로그람은 이 사건에 응답한 후 상태띠에서 부품용기를 제거하였다는 정보를 현시한다.

Applet는 일종의 특수한 panel이며 Java Applet프로그람의 제일 바깥층용기이다. 그러나 JavaApplet는 완전한 독립적인 프로그람이 아니며 실제상 WWW열람기에서의 하나의 조종부품이다. 열람기의 한 부분으로서 열람기에 의존하여 존재하므로 Java Applet는 열람기의 창문에 따라 확대, 축소, 닫기 등의 기능을 완성할수 있다.

Applet용기의 기정배치방안은 그의 상위클라스와 일치하며 모두 FlowLayout이다. 그러나 Applet용기에서는 열람기와 서로 작용하는 메쏘드를 객관적으로 정의하고있다.(례: init( ), start( ), stop( ) 등)

제11절. Frame과 창문사건

- **Frame은 가장 많이 사용되는 테두리있는 용기이다.**
- **구성자: Frame( ), Frame(String title)**
- **WindowsEvent의 주요메쏘드: public window getWindow( )**

Container는 Applet, Panel과 같은 테두리없는 용기뿐아니라 테두리있는 용기인 Window, Frame, Dialog, FileDialog들도 포함된다. 여기서 Window는 모든 테두리있는 용기들의 상위클라스이다. 그러므로 이 안의 다른 용기는 모두 테두리를 가지고 독립적으로 존재할수 있다.

8.11.1. Frame

앞의 례제에서 Frame용기를 이미 사용한적이 있다. Frame은 Application의 제일 바깥층 용기로 될수도 있고 다른 용기에 의하여 창조되여 독립적인 용기로 리용될수도 있다. 그러나 어떤 경우일지라도 Frame은 제일 웃층의 용기로 존재하지 다른 용기에 포함될수 없다.

Frame은 자기의 바깥테두리와 표제를 가지며 Frame창조시 이 창문표제를 지정할수 있다.

Frame(String title);

getTitle( )과 setTile(string)메쏘드를 사용하여 Frame의 표제를 얻거나 지정할수도 있다. 새로 창조하는 Frame은 볼수 없으며 setVisible(boolean)메쏘드를 사용하여 실제파라메터를 true로 하여야 볼수 있다.

매개 Frame은 오른쪽웃구석에 3개의 조종아이콘을 가지며 그것들은 각각 창문의 최소화, 최대화, 닫기조작을 실현한다. 여기서 최소화와 최대화는 Frame을 조작하여 자동적으로 완성할수 있다. 창문크기의 조작은 아이콘을 찰각하여 실현할수 없으며 프로그람의 전문적인 편관코드를 작성하여야 한다. 보통 리용하는 창문닫기의 메소드에는 3가지가 있다. 하나는 단추를 설치하고 사용자가 단추를 찰각할 때 창문을 닫는것이다. 두번째 메소드는 WINDOWS.CROSING사건에 응답하여 창문을 닫는것이며 세번째 메소드는 차림표명령을 사용하는것이다. 첫번째 메소드는 전문적인 단추를 요구하며 두번째 메소드는 WindowsListener대면을 실현하는 코드를 작성해주어야 한다. 어느 메소드를 사용하더라도 Frame을 닫는 dispose()메소드를 사용하여야 한다.

Frame창문에 부품을 추가하고 제거하자면 다른 용기들과 마찬가지로 add( )메소드와 remove()메소드를 사용하여야 한다. Frame은 WindowsEvent클라스가 나타내는 7가지 창문사건을 일으킬수 있다.

8.11.2. 창문사건

WindowsEvent클라스는 아래와 같은 구체적인 창문사건을 가지고있다.

① WINDOW_ACTIVATED: 창문이 능동으로 되였다는것을 의미한다.(화면의 제일 앞에서 명령을 기다린다.)

② WINDOW_DEACTIVATED: 창문의 비능동상태를 의미한다.(다른 창문이 능동으로 되면 원래의 능동창문은 비능동상태로 된다.)

③ WINDOW_OPENED: 창문이 열려졌다는것을 의미한다.

④ WINDOW_CLOSED: 창문이 이미 닫기였다는것을 의미한다.

⑤ WINDOW_CLOSING: 창문이 현재 닫기고있다는것을 의미한다.

⑥ WINDOW_ICONIFIED: 아이콘으로 창문을 최소화하는것을 의미한다.

⑦ WINDOW_DEICONIFIED: 창문이 아이콘으로부터 복귀되였다는것을 의미한다.

WindowsEvent클라스의 주요메소드에는 getWindow()가 있다. 이 메소드는 현재 WindowsEvent사건을 일으키는 구체적인 창문을 귀환시키며 getSource( )메소드가 귀환시키는것과 같은 사건이 인용된다. 그러나 getSource( )의 귀환값은 Object이며 getWindow()메소드의 귀환값은 구체적인 Window객체이다.

실례 8-20

```
Example 8-20 TestFrame.java
1: import java.awt.*;
2: import java.awt.event.*;
3:
4: public class TestFrame
5: {
```

```
6:      public static void main(String args[])
7:      {
8:          new MyFrame( );
9:      }
10:}
11:class MyFrame extends Frame implements ActionListener
12:{
13:     Button btn;
14:     MyFrame( )
15:     {
16:         super("My Window");
17:         btn = new Button("닫기");
18:         setLayout(new FlowLayout( ));
19:         add(btn);
20:         btn.addActionListener(this);
21:         addWindowListener(new closeWin( ));
22:         setSize(300, 200);
23:         setVisible(true);
24:     }
25:     public void actionPerformed(ActionEvent e)
26:     {
27:         if(e.getActionCommand( ) == "닫기")
28:         {
29:             dispose( );
30:             System.exit(0);
31:         }
32:     }
33:}
34:class closeWin extends WindowAdapter
35:{
36:     public void windowClosing(WindowEvent e)
37:     {
38:         Frame frm = (Frame)(e.getSource( ));
39:         frm.dispose( );
40:         System.exit(0);
41:     }
42:}
```

그림 8-21은 실례 8-20의 실행결과이다.

그림 8-21. 실례 8-20의 실행결과

프로그람설명

이것은 도형대면의 Java Application프로그람이다. 프로그람에서는 3개의 클라스를 정의하고있다. closeWin은 창문사건의 오림클라스 WindowAdapter의 하위클라스이며 여기서 WindowClosing( )메쏘드를 재정의하여 사용자가 닫기아이콘단추를 찰칵할 때 사건원천창문을 닫는다. MyFrame은 사용자가 정의한 체계클라스 Frame의 하위클라스이며 안에 한개의 단추를 포함한다.

제12절. 차림표의 사용

- 차림표는 부품클라스로부터 계승되지 않는다.
- MenuBar, Menu, MenuItem, CheckboxMenuItem클라스로 구성된다.
- Frame클라스의 setMenuBar(MenuBar m)로 MenuBar를 추가한다.

Frame용기에 대하여 특별히 론의해야 할 점은 Frame이 차림표의 용기를 가질수 있으며 그것은 MenuContainer대면을 실현하고있다는것이다. 차림표는 아주 중요한 GUI부품으로서 매개 차림표부품은 MenuBar라고 하는 한개의 차림표띠를 가지며 차림표띠는 또 Menu라고 하는 몇개의 차림표항목을 가진다. 그리고 매개 차림표항목은 MenuItem이라고 하는 몇개의 차림표부분항목을 포함한다. 매개 차림표부분항목의 작용은 단추와 마찬가지로 사용자가 찰칵할 때 한개의 동작명령을 일으키므로 전체 차림표는 계층화되여 조직관리되는 하나의 명령모임이다. 그것을 사용하면 사용자는 쉽게 프로그람을 작성할수 있다.(실례 8-21)

실례 8-21

```
Example 8-21 TestMenu.java
1: import java.awt.*;
2: import java.awt.event.*;
3:
4: public class TestMenu
5: {
6:      public static void main(String args[])
```

```
7:      {
8:          MyMenuFrame mf = new MyMenuFrame( );
9:          mf.setSize(new Dimension(300, 200)); //창문크기설정
10:         mf.setVisible(true);
11:     }
12:}
13:
14:class MyMenuFrame extends Frame implements ActionListener,
                                        ItemListener   //창문정의
15:{
16:     MenuBar m_MenuBar;
17:     Menu menuFile, menuEdit, m_Edit_Paste;
18:     MenuItem mi_File_Open, mi_File_Close, mi_File_Exit, mi_Edit_Copy;
19:     MenuItem pi_New, pi_Del, pi_Pro, mi_Paste_All, mi_Paste_Part;
20:     CheckboxMenuItem mi_Edit_Cut;
21:     PopupMenu popM;
22:     TextArea ta;
23:
24:     MyMenuFrame( )
25:     {
26:         super("Window with a menu");
27:         ta = new TextArea("\n\n\n\n\n\n\t\t\t선택하지  않음", 5, 20);
28:         ta.addMouseListener(new HandleMouse(this));
                            //본문구역이 마우스사건에 응답
29:         add("Center", ta);
30:
31:         popM = new PopupMenu( );
32:         pi_New = new MenuItem("창조");
33:         pi_New.addActionListener(this);
34:         popM.add(pi_New);
35:         pi_Del = new MenuItem("삭제");
36:         pi_Del.addActionListener(this);
37:         popM.add(pi_Del);
38:         pi_Pro = new MenuItem("속성");
39:         pi_Pro.addActionListener(this);
40:         popM.add(pi_Pro);
41:         ta.add(popM);
42:
43:         m_MenuBar = new MenuBar( ); //차림표띠창조
44:
45:         menuFile = new Menu("파일");
46:         mi_File_Open = new MenuItem("열기", new MenuShortcut('O'));
```

```
47:         mi_File_Close = new MenuItem("닫기");
48:         mi_File_Exit = new MenuItem("탈퇴");
49:         mi_File_Exit.setShortcut(new MenuShortcut('X'));
50:         mi_File_Open.setActionCommand("열기");
51:         mi_File_Close.setActionCommand("닫기");
52:         mi_File_Open.addActionListener(this);
53:         mi_File_Close.addActionListener(this);
54:         mi_File_Exit.addActionListener(this);
55:         menuFile.add(mi_File_Open);
56:         menuFile.add(mi_File_Close);
57:         menuFile.addSeparator( );
58:         menuFile.add(mi_File_Exit);
59:         m_MenuBar.add(menuFile);
60:
61:         menuEdit = new Menu("편집");
62:         mi_Edit_Copy = new MenuItem("복사");
63:         mi_Edit_Cut = new CheckboxMenuItem("자르기");
64:         m_Edit_Paste = new Menu("붙이기");
65:         mi_Paste_All = new MenuItem("모두 붙이기");
66:         mi_Paste_Part = new MenuItem("부분 붙이기");
67:         mi_Edit_Copy.addActionListener(this);
68:         mi_Edit_Cut.addItemListener(this);
69:         m_Edit_Paste.add(mi_Paste_Part);
70:         m_Edit_Paste.add(mi_Paste_All);
71:         mi_Paste_Part.addActionListener(this);
72:         mi_Paste_All.addActionListener(this);
73:         menuEdit.add(mi_Edit_Copy);
74:         menuEdit.add(mi_Edit_Cut);
75:         menuEdit.addSeparator( );
76:         menuEdit.add(m_Edit_Paste);
77:         m_MenuBar.add(menuEdit);
78:
79:         this.setMenuBar(m_MenuBar); //차림표띠를 Frame용기에 추가
80:     }
81:     public void actionPerformed(ActionEvent e)
82:     {
83:         if(e.getActionCommand( ) == "탈퇴")
84:         {
85:             dispose( );
86:             System.exit(0);
87:         }
88:         else
```

```
89:             ta.setText("\n\n\n\n\n\n\t\t\t"+e.getActionCommand( ));
90:     }
91:     public void itemStateChanged(ItemEvent e)
92:     {
93:         if(e.getSource( ) == mi_Edit_Cut)
94:             if(((CheckboxMenuItem)e.getSource( )).getState( ))
95:                 ta.setText("\n\n\n\n\n\n\t\t\t" +
96:                 ((CheckboxMenuItem)e.getSource( )).getLabel( ) +
97:                  를 선택하였습니다.");
98:             else
99:                 ta.setText("\n\n\n\n\n\n\t\t\t" +
100:                   ((CheckboxMenuItem)e.getSource( )).getLabel( ) +
101:                   "를 선택하지 못하였습니다.");
102:    }
103:}
104:
105: class HandleMouse extends MouseAdapter      //마우스사건클라스의 처리
106: {
107:     MyMenuFrame m_Parent;
108:
109:     HandleMouse(MyMenuFrame mf)
110:     {
111:         m_Parent = mf;
112:     }
113:     public void mouseReleased(MouseEvent e)
114:     {
115:         if(e.isPopupTrigger( ))
116:             m_Parent.popM.show((Component)e.getSource( ), e.getX( ), e.getY( ));
117:     }
118: }
```

프로그람설명

실례 8-21은 Java에서 제공하는 차림표의 기능들을 사용하고있다. Java에서의 차림표는 2가지로 갈라진다. 즉 하나는 차림표띠형식의 차림표로서 보통 우리가 말하는 차림표를 말한다. 다른 하나는 튀여나오기차림표이다. 아래에서 우선 차림표띠형식의 차림표에 대하여 고찰한다.

8.12.1. 차림표의 설계와 실현

차림표의 설계와 실현단계는 아래와 같다.

・ 차림표띠 MenuBar를 창조한다. 아래의 명령문은 빈 차림표띠를 창조한다.

```
m_MenuBar = new MenuBar( );
```

・ 서로 다른 차림표항목 Menu를 창조하여 빈 차림표띠에 추가한다. 아래의 명령문은 《편집》이라는 차림표항목을 창조하고 그것을 차림표띠에 추가한다.

```
menuEdit = new Menu("편집");
m_MenuBar.add(menuEdit);
```

・ 매개 차림표항목에 대하여 그것에 포함된 차림표부분항목 MenuItem들을 창조하고 차림표부분항목을 차림표항목에 추가한다.

```
mi_Edit_Copy = new MenuItem("복사");
menuEdit.add(mi_Edit_Copy);
```

・ 구축된 전체 차림표를 어떤 용기에 추가한다.

```
this.setMenuBar(m_MenuBar);
```

여기서 this는 프로그람의 용기 Frame을 의미하며 주의할것은 MenuContainer대면을 실현한 용기에만 차림표띠형식의 차림표를 추가할수 있다는것이다.

・ 매 차림표부분항목을 조작사건의 감시자대면 ActionListener를 실현한 감시자에게 등록하여야 한다.

```
mi_Edit_Copy.addActionListener(this);
```

・ 감시자에 대하여 actionPerformed(Action Event e)메쏘드를 정의한다. 이 메쏘드에서 e.getSource( )나 e.getActionCommand( )를 호출하여 사용자가 찰칵하는 차림표부분항목을 판단하고 이 부분항목이 정의하는 조작이 서술된 표제를 귀환시킨다.

창조한 차림표띠형식의 차림표는 창문의 웃쪽에 놓인다. 그림 8-22는 실례 8-21에서 창조한 차림표띠를 현시한다.

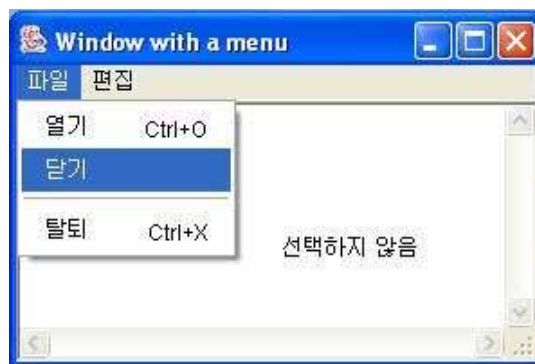



그림 8-22. 실례 8-21의 실행결과

8.12.2. 분리선사용

차림표부분항목사이에 가로방향의 분리선을 추가하여 차림표부분항목을 몇개로 갈라야 할 때가 있다. 분리선을 추가하는 메쏘드는 addSeparator( )이다. 사용시 이 명령문의 위치에 주의하여야 한다. 차림표부분항목은 추가한 선후차에 따라 차림표항목에 배렬되며 분리선을 넣고싶은 곳에 분리선명령문을 놓는다. 실례로 《파일》차림표항목에서 아래의 명령문을 사용하여 분리선을 추가하였다(그림 8-22).

```
menuFile.addSeparator( ); // 가로방향분리선을 추가
```

8.12.3. 지름건사용

마우스를 리용하여 차림표부분항목을 선택하는외에 매개 차림표부분항목에 대하여 건반지름건을 정의하고 건반을 사용하여 차림표부분단추를 선택할수 있다. 지름건은 하나의 자모로서 정의한 후에 ctrl건과 이 자모를 누르면 대응하는 차림표부분항목을 선택할수 있다. 차림표부분항목에 대하여 지름건을 정의하는 메쏘드에는 2가지가 있다. 하나는 차림표부분항목을 창조하는 동시에 지름건을 정의하는것이다.

```
MenuItem mi_File_Open = new MenuItem("열기", new MenuShortcut('O'));
```

다른 하나는 이미 존재하는 차림표부분항목에 지름건을 정의하는것이다.

```
mi_File_Exit.setShortcut(new MenuShortcut('X'));
                                        //차림표부분항목의 지름건을 단독설정
```

설치후에 차림표부분항목 《열기》는 지름건 ctrl+O에 대응하며 《탈퇴》는 ctrl+X에 대응한다.(그림 8-22)

8.12.4. 2준위차림표사용

만일 앞으로 차림표부분항목들을 차림표항목에 더 많이 추가하려면 2준위차림표를 리용하여 간단히 실현할수 있다. 2준위차림표의 사용방법은 아주 간단하며 몇개의 차림표부분항목(MenuItem)들을 가지는 차림표항목(Menu)을 창조한다. 이 차림표항목을 차림표부분항목과 같이 1준위차림표항목에 추가할수 있다.

```
m_Edit_Paste = new Menu("붙이기"); // 2준위차림표항목창조
mi_Paste_All = new MenuItem("전부 붙이기");
mi_Paste_Part = new MenuItem("부분 붙이기");
m_Edit_Paste.add(mi_Paste_Part); // 2준위차림표항목에 차림표부분항목을 추가
m_Edit_Paste.add(mi_Paste_All);
menuEdit.add(m_Edit_Paste); // 2준위차림표항목을 차림표항목에 추가
```

2준위차림표의 사용효과를 그림 8-23에 보여준다.

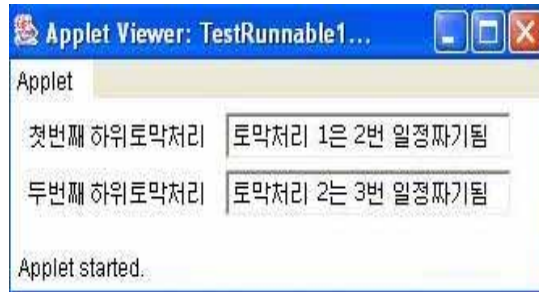



그림 8-23. 2준위차림표의 사용효과

8.12.5. 검사칸차림표부분항목의 사용

Java에서는 또한 검사칸차림표부분항목 CheckboxMenuItem이라는 일종의 특수한 차림표부분항목을 정의하고있다. 이 차림표부분항목은 검사칸과 같으며 《선택》과 《비선택》의 두상태가 있다. 매번 차림표부분항목의 선택은 두 상태사이를 절환하며 선택상태에 있는 검사칸차림표부분항목의 앞에는 작은 대조기호(그림 8-23)가 생기고 비선택상태에 놓일 때에는 이 대조기호가 없어진다. 검사칸차림표부분항목을 창조하고 그것을 차림표항목의 메쏘드에 아래와 같이 추가한다.

```
mi_Edit_Cut = new CheckboxMenuItem("자르기");
menuEdit.add(mi_Edit_cut);
```

검사칸차림표부분항목을 선택하여 일으키는 사건은 동작사건 ActionEvent가 아니라 선택사건 ItemEvent이므로 검사칸차림표부분항목을 ItemListener에 등록해야 하며 ItemListener의 ItemStateChanged(ItemEvente)사건을 구체적으로 실현하여야 한다. 이것은 검사칸에 응답하는 사건과 류사하다.

```
mi_Edit_Cut.addItemListener(this);
```

8.12.6. 튀여나오기차림표의 사용

튀여나오기차림표는 어떤 부품이나 용기에 부속되여 일반적으로 그것을 볼수 없고 사용자가 마우스의 오른쪽건을 사용하여 튀여나오기차림표의 부품을 창조할 때에만 이 차림표가 현시된다.

튀여나오기차림표는 차림표띠형식의 차림표와 같이 몇개 차림표부분항목을 포함하는데 먼저 튀여나오기차림표를 창조하고 차림표부분항목을 아래와 같이 추가하여야 한다.

```
ta.add(popM); //튀여나오기창문을 본문구역에 추가
```

다음에 튀여나오기차림표를 어떤 부품이나 용기에 부속시켜야 한다. 사용자가 마우스의 오른쪽건을 찰칵할 때 튀여나오기차림표는 자동적으로 현시되여 일정한 프로그람처리를 요구한다. 차림표를 튀여나오게 하는 부품이나 용기를 MouseListener에 등록하여야 한다.

```
ta.addMouseListener(new HandleMouse(this));
                    //본문구역은 마우스사건에 응답하고 차림표를 튀여나오게 한다.
```

다음 MouseListener의 mouseReleased(MouseEvent e)메쏘드를 재정의한다. 이 메쏘드에서 튀여나오기차림표의 show( )메쏘드를 호출하여 그 자체를 사용자가 마우스를 찰칵하는 위치에 현시한다.

```
public void mouseReleased(MouseEvent e)
     //마우스건을 눌러 사건을 해방시키고 차림표를 튀여나오게 한다.
{
     if(e.isPopupTrigger( ))
      //마우스사건이 튀여나오기차림표에 의해 일어나는가를 검사
     m_Parent.popM.show((Component)e.getSource(),e.getX(),e.getY());
}    //튀여나오기차림표를 사용자가 마우스를 한번 찰칵하는 위치에 현시
```

여기서 메쏘드 e.getSourse( )가 귀환하는것은 튀여나오기차림표가 부속된 부품이나 용기이며 튀여나오기차림표는 이 부품이나 용기에서 마우스가 찰칵하는 위치에 현시되여야 한다.(e.getX( )와 e.getY( )메쏘드는 마우스를 찰칵하는 자리표위치를 확정한다.)(그림 8-24)

그림 8-24. 튀여나오기차림표의 현시와 사용

제13절. 대화칸, 부품사건과 초점사건

- Dialog는 반드시 Frame이나 다른 Dialog에 속해야 한다.
- Modal 또는 Modaless상태표현이 가능하다.
- 구성자:
  Dialog(Frame owner, String title, boolean modal)
  Dialog(Dialog owner, String title, boolean modal)
- ComponentEvent와 FocusEvent의 사건들은 getID( )메쏘드를 호출하여 비교한다.

8.13.1. 부품사건

이 클라스는 낮은 수준사건의 뿌리클라스이며 모두 4개의 구체적사건을 포함하고 있다. 이것들은 componentEvent클라스의 몇개 정적상수를 리용하여 표시한다.

· componentEvent.COMPONENT_HIDEN: 부품을 감추는 사건을 의미한다.

· componentEvent.COMPONENT_SHOWN: 부품을 현시하는 사건을 의미한다.

· componentEvent.COMPONENT_MOVED: 부품을 이동시키는 사건을 의미한다.

· componentEvent.COMPONENT_RESIZED: 부품의 크기를 변동시키는 사건을 의미한다.

getID( )메쏘드의 귀환값을 우의 상수들과 서로 비교하면 componentEvent객체가 나타내는 구체적인 사건을 알수 있다.

8.13.2. 초점사건

FocusEvent클라스는 2개의 사건을 포함한다. 사건들은 이 클라스와 이름이 같은 두개의 정적옹근수상수에 각각 대응한다.

FOCUS_GAINED: 초점얻기를 의미한다.

FOCUS_LOST: 초점을 잃는다는것을 의미한다.

GUI객체는 반드시 초점을 얻어야 조작할수 있다. 실례로 하나의 본문입력구역은 우선 초점을 얻어야 사용자가 입력한 문자를 접수할수 있으며 창문도 초점을 얻어야 차림표를 선택할수 있다. 초점을 얻은 객체는 전체 화면의 제일 앞에서 명령을 기다리는 상태에 있는 기정조작의 모든 객체이다. 초점을 잃은 객체는 화면의 뒤에로 옮겨가 다른 객체를 막을수 있다.

8.13.3. 대화칸

Frame과 마찬가지로 Dialog는 테두리와 표제를 가지는 독립적인 용기로서 다른 용기에 포함될수 없다. 또한 Dialog는 프로그람의 제일 바깥층 용기로 될수도 없고 차림표띠를 포함할수도 없다. Window와 같이 Dialog는 반드시 하나의 Frame에 속해야 하며 이 Frame으로부터 철저히 튀여나와야 한다.

Dialog는 보통 사용자와 서로 작용하는 대화칸의 역할을 한다. 실례로 사용자에게 통보문을 전하고 확인된 통보문을 요구하며 사용자가 입력한 일반대화칸문을 접수한다. Dialog의 구성자는 4가지 재정의방식을 가지며 여기서 가장 복잡한것은 Dialog(frame parent, String title, boolean isModal)이다. 첫번째 파라메터는 새로 창조하는 Dialog대화칸이 어느 Frame창문에 속하는가를 가리키며 두번째 파라메터는 새로 창조하는 Dialog대화칸의 표제를 가리킨다. 세번째 파라메터는 이 대화칸이 양상(modal)을 가지는가를 지정한다. 《양상을 가지는》 대화칸이란 일단 열기한 후 사용자가 반드시 이에 응답하여야 하는 대화칸이다. 레를 들어 대화칸은 사용자가 삭제조작을 확인하였는가를 문의한다. 이때 프로그람은 림시정지상태에 놓이며 사용자가 대화칸의 질문에 대답하지 않으면 프로그람의 다른 부품을 사용할수 없다. 그러므로 강제적인 성질을 가지고있다. 한편 양상이 없는 대화칸은 이러한 제한이 없으며 사용자는 이 대화칸을 열기한 후에도 프로그람의 다른 부분을 조작할수 있다. 일반적으로 기정인 상태에서 대화칸은 양상이 없는것이다. 새로 창조하는 대화칸은 기정의 BorderLayout를 사용하므로 show( )메쏘드를 사용하여 그것을 현시할수 있다.

이미 창조한 대화칸에 대하여 setModal(booleam isModal)메쏘드를 리용하여 이 양상속성을 변경할수 있으며 또는 boolean isModal( )메쏘드를 사용하여 그것이 양상을 가지는 대화칸인가를 판단할수 있다. 그밖에 Dialog는 대화칸의 표제를 얻고 수정하는 메쏘드 getTitle( )과 setTitle(string newTitle)을 가지며 부품을 추가하고 제거하는 메쏘드 add(component)와 remove(component)를 가지고있다.

Dialog를 리용하여 실현한 통보문대화칸과 일반대화칸의 실례를 고찰한다.

실례 8-22

```
Example 8-22 TestDialog.java
1:  import java.awt.*;
2:  import java.awt.event.*;
3:
4:  public class TestDialog
5:  {
6:      public static void main(String args[])
7:      {
8:          MyDialogFrame df = new MyDialogFrame( );
9:      }
```

```
10: }
11:
12: class MyDialogFrame extends Frame implements
                              ActionListener, ComponentListener, FocusListener
13: {
14:     Dialog MegDlg, InOutDlg; //대화칸은 Frame에 속한다.
15:     Button btn1, btn2, btnY, btnN, btnR;
16:     TextField tf = new TextField("정보가 없음", 45);
17:     TextField getMeg = new TextField("정보입력", 10);
18:
19:     MyDialogFrame( )
20:     {
21:         super("Using Dialog");
22:         btn1 = new Button("감추기");
23:         btn2 = new Button("조사");
24:         btnY = new Button("예");
25:         btnN = new Button("아니");
26:         btnR = new Button("되돌이");
27:         setLayout(new FlowLayout( ));
28:         add(tf);
29:         add(btn1);
30:         add(btn2);
31:         btn1.addComponentListener(this);
32:         this.addWindowListener(new WinAdpt( ));
33:         btn1.addActionListener(this);
34:         btn2.addActionListener(this);
35:         btnY.addActionListener(this);
36:         btnN.addActionListener(this);
37:         btnR.addActionListener(this);
38:         setSize(350, 150);    //크기변경
39:         show( );   //현시
40:     }
41:     public void actionPerformed(ActionEvent e)
42:     {
43:         if(e.getActionCommand( ) == "감추기")
44:         {
45:             MegDlg = new Dialog(this, "Hiding", true);
46:             Panel pl = new Panel( );
47:             pl.add(new Label("조작은 이 단추를 감추기합니다. 계속하시렵니까?"));
```

```
48:             MegDlg.add("Center", pl);
49:             Panel p2 = new Panel( );
50:             p2.add(btnY);
51:             p2.add(btnN);
52:             MegDlg.add("South", p2);
53:             MegDlg.setSize(200, 100);
54:             MegDlg.show( );   //대화칸현시
55:         }
56:         else if(e.getActionCommand( ) == "조사")
57:         {
58:             InOutDlg = new Dialog(this, "Input a Information");
59:             InOutDlg.add("Center", getMeg);
60:             InOutDlg.add("South", btnR);
61:             InOutDlg.setSize(200, 100);
62:             InOutDlg.addFocusListener(this);
63:             InOutDlg.show( );
64:         }
65:         else if(e.getActionCommand( ) == "예")
66:         {
67:             MegDlg.dispose( );
68:             btn1.setVisible(false);
69:         }
70:         else if(e.getActionCommand( ) == "아니")
71:             MegDlg.dispose( );
72:         else if(e.getActionCommand( ) == "되돌이")
73:         {
74:             tf.setText(getMeg.getText( ) + "은 대화칸의 입력입니다");
75:             InOutDlg.dispose( );
76:         }
77:     }
78:     public void componentShown(ComponentEvent e){ }
79:     public void componentResized(ComponentEvent e){ }
80:     public void componentMoved(ComponentEvent e){ }
81:     public void componentHidden(ComponentEvent e)
82:     {
83:         tf.setText("단추\"" + ((Button)e.getComponent( )).getLabel( )
                                                    + "\"감추어졌습니다!");
84:     }
85:     public void focusGained(FocusEvent e)
```

```
86:     {
87:         getMeg.setText("대화칸\"" + ((Dialog)e.getComponent( )).getTitle( )
                                              + "\"은 주의초점을 얻었습니다!");
88:     }
89:     public void focusLost(FocusEvent e){ }
90: }
91: class WinAdpt extends WindowAdapter //창문오림클라스창조
92: {
93:     public void windowClosing(WindowEvent e)
94:     {
95:         ((Frame)e.getWindow( )).dispose( );
96:         System.exit(0);
97:     }
98: }
```

프로그람설명

이 프로그람의 제일 바깥층의 용기는 Frame창문으로서 그것은 동작사건과 용기사건에 응답할수 있다. 창문에는 한개의 본문칸과 두개의 단추가 있다. 사용자가 첫번째 단추 《감추기》를 찰칵할 때 통보문대화칸 MegDlg가 튀여나온다. 여기에 한개의 Label이 있는데 이것은 사용자가《조작은 이 단추를 감추기합니다. 계속하시렵니까?》를 문의한다. 이밖에 두개의 단추 《예》와 《아니》가 있다. 만일 《예》를 누르면 대화칸을 닫고 《감추기》단추를 감추며 이때 단추감추기조작은 부품사건을 일으켜 련관정보를 Frame본문칸 tf에 현시한다.

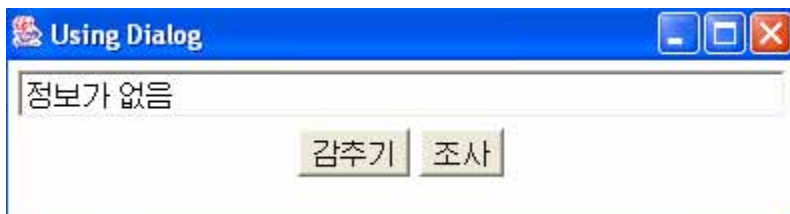
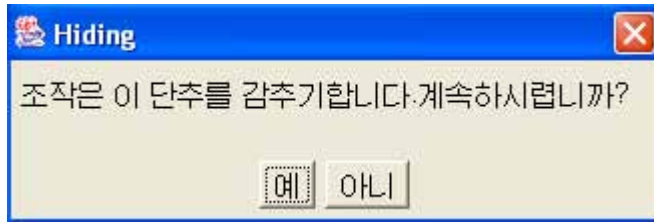
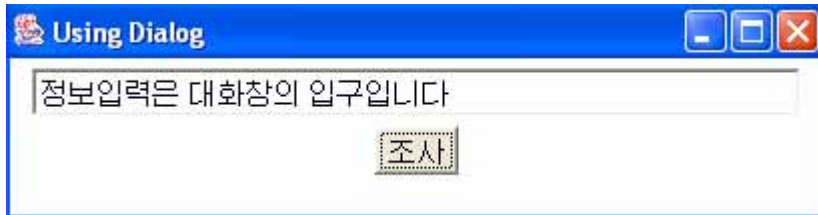
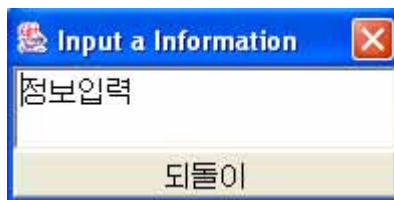
사용자가 Frame의 두번째 단추 《조사》를 찰칵할 때는 입출력대화칸 InOutDlg가 튀여나오며 사용자는 이 대화칸의 본문칸에 정보를 입력한다. InOutDlg대화칸은 주의초점을 얻는 사건에 응답할수 있으며 사용자가 입력을 끝낸 후 《되돌이》단추를 찰칵할 때 대화칸을 닫고 사용자가 입혁한 본문을 본문칸 tf에 보내여 현시한다. 프로그람결과는 그림 8-25에서 보여준다.

(ㄱ) 초기의 Frame

(ㄴ) 양상이 있는 대화칸

(ㄷ) 양상이 없는 대화칸

(ㄹ) 대화칸으로부터 정보의 얻기

그림 8-25. 실례 8-22의 실행결과

Dialog는 또한 하위클라스 FileDialog를 가지고있는데 이것은 특정한 등록부와 파일을 탐색하는데 쓰인다. 특정한 파일을 열거나 보관하는 대화칸은 Java의 파일조작에 대하여 취급할 때 소개한다.

제14절. Swing GUI부품

GUI부품: Japplet, JButton

Java에서 java.swing패키지는 Java의 기초클라스서고(JFC)에 포함되여있다. 여기서 정의하는 Swing GUI부품은 Java.awt패키지의 각종 GUI부품에 상응하게 많은 기능을 추가하였으며 이 절에서는 몇가지만을 고찰한다.

8.14.1. JApplet

javax.swing.JApplet는 java.applet.Applet의 하위클라스이며 모든 Swing GUI 부품은 JApplet프로그람에 포함되여야 한다. 실례 8-23은 Japplet를 실현한 간단한 례제로서 이 사용방법은 Applet프로그람과 비슷하다.

실례 8-23

```
Example 8-23 TestJApplet.java
1: import javax.swing.*;
2: import java.awt.*;
3:
4: public class MyFirstJApplet extends JApplet
5: {
6:     public void paint(Graphics g)
7:     {
8:         g.drawString("이것은 JApplet프로그람입니다", 10, 20);
9:     }
10:}
```

JApplet프로그람과 배합하여 사용하는 HTML파일과 Applet프로그람과 배합하여 사용하는 HTML파일은 아무런 차이가 없다.

그림 8-26은 실례 8-23의 실행결과이다.

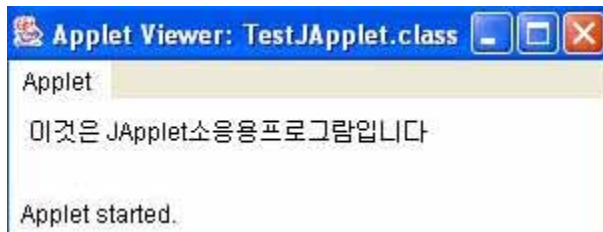



그림 8-26. 실례 8-23의 실행결과

JApplet와 Applet의 차이는 Applet의 기정배치방안은 FlowLayout이고 JApplet의 기정배치방안은 BorderLayout라는데 있다. 그밖에 JApplet에 Swing부품을 추가할 때 add( )메쏘드를 직접 리용할수 없으며 반드시 먼저 JApplet의 getContentPane( )메쏘드를 사용하여 Container객체를 얻어야 한다. 다시 이 Container객체의 add( )메쏘드를 호출하여 JComponent와 그의 하위클라스객체를 JApplet에 추가한다.

JComponent클라스는 Swing GUI부품의 모든 상위클라스이며 이 Swing GUI부품들은 JApplet프로그람에 추가되든가 Frame용기에 추가될수 있다. 기본적으로 매개 java.awt부품에 대응하며 javax.swing의 《J부품》에 존재한다. 실례로 Button에 대응하는것은 JButton이고 Label에 대응하는것은 JLabel이다. 어떤 《J부품》은 대응하는 AWT부품의 기능 및 작용과 류사하며 어떤것은 크게 갱신되였다.

8.14.2. JButton

Button클라스와 같이 JButton클라스는 많은 실용적인 기능을 새로 추가하였다. 실례로 Swing단추에서 아이콘을 현시하고 서로 다른 상태에서 서로 다른 Swing단추아이콘을 사용한다. Swing단추에 대한 정보 등을 추가제시한다.

1) **단추아이콘창조**

JButton객체는 Button객체와 같이 파일표식자를 리용할수 있는것외에 하나의 아이콘을 쓸수 있다. 이 아이콘은 사용자자체가 그린 도형일수도 있고 이미 존재하는 .gif화상일수도 있다.

실례 8-24

```
Example 8-24 TestIconButton.java
1: import java.awt.*;
2: import java.awt.event.*;
3: import javax.swing.*;
4:
5: public class TestIconButton extends JApplet implements ActionListener
6: {
7:       JButton jbtn;
8:       public void init( )
9:       {
10:           Container c = getContentPane( );
11:           Icon icon = new ImageIcon("bIcon.gif");
12:           jbtn = new JButton("J Btn", icon);
13:           c.add(jbtn, BorderLayout.NORTH);
14:           jbtn.addActionListener(this);
```

```
15:     }
16:     public void actionPerformed(ActionEvent e)
17:     {
18:         showStatus("J Btn의 찰칵에 응답");
19:     }
20:}
```

그림 8-27은 실례 8-24의 실행결과이다.

그림 8-27. 실례 8-24의 실행결과

프로그람설명

실례 8-24에서 10행은 getContentPane( )메쏘드를 호출하여 Container객체를 얻고 JApplet에 Swing부품을 추가하는데 리용하였다.

11행은 icon을 창조하고있다. 12행은 이 아이콘을 JButton객체 jbtn에 추가하였으며 이 Swing단추는 동시에 문자표식자 《J Btn》을 사용하였다. 13행은 이 JButton객체를 JApplet에 추가하였다. 실례 8-24로부터 알수 있는것처럼 JButton의 사건응답은 Button과 완전히 같다.

2) 단추아이콘변경

JButton단추는 한개의 아이콘을 리용할수도 있고 한개이상의 아이콘도 리용할수 있다. 또한 Swing단추가 처리하는 내용에 따라 Swing단추아이콘을 자동변환시킬수도 있다.

실례 8-25

```
Example 8-25 TestChangedIcon.java
1: import java.awt.*;
2: import java.awt.event.*;
3: import javax.swing.*;
4:
```

```
5: public class TestChangedIcon extends JApplet implements ActionListener
6:  {
7:     JButton jbtn;
8:     public void init( )
9:     {
10:        Container c = getContentPane( );
11:        Icon normalIcon = new ImageIcon("join_d.gif");
12:        Icon pressedIcon = new ImageIcon("join_up.gif");
13:        Icon rolloverIcon = new ImageIcon("join_d.gif");
14:        jbtn = new JButton(normalIcon);
15:        jbtn.setPressedIcon(pressedIcon);
16:        jbtn.setRolloverIcon(rolloverIcon);
17:        jbtn.setRolloverEnabled(true);
18:        c.add(jbtn, BorderLayout.NORTH);
19:        jbtn.addActionListener(this);
20:    }
21:    public void actionPerformed(ActionEvent e)
22:    {
23:        showStatus("J Btn의 한번 찰칵에 응답");
24:    }
25: }
```

그림 8-28은 실례 8-25의 실행결과이다.

그림 8-28. 실례 8-25의 실행결과

프로그람설명

실례 8-25에서 11-13행은 3개의 아이콘을 창조하고있다. 14행은 아이콘만 있고 문자표식자가 없는 한개의 Swing단추를 창조한다. 이 아이콘은 Swing단추가 늘 현시하는 아이콘이다. 15행에서 Swing단추가 눌리워질 때 현시하는 아이콘을 두번째 아이콘으로 설정한다. Swing단추의 이 기능은 프로그람작성자가 도형사용자대면부설계를 보다 편리하게 할수 있게 한다.

3) 단추에 입력재촉정보를 추가

실제적으로 사용하는 많은 단추들에 하나의 기능을 추가할수 있는데 그것이 바로 마우스가 단추에 몇초동안 머무르고있을 때 화면상에서 이 단추를 설명하는 짧은 입력재촉정보를 현시할수 있는것이다.

실례 8-26

```
Example 8-26 TestTipButton.java
1: import java.awt.*;
2: import java.awt.event.*;
3: import javax.swing.*;
4:
5: public class TestTipButton extends JApplet implements ActionListener
6: {
7:     JButton jbtn;
8:     public void init( )
9:     {
10:         Container c = getContentPane( );
11:         Icon icon = new ImageIcon("www.gif");
12:         jbtn = new JButton(icon);
13:         jbtn.setToolTipText("This a Swing button!");
14:         c.add(jbtn, BorderLayout.NORTH);
15:         jbtn.addActionListener(this);
16:     }
17:     public void actionPerformed(ActionEvent e)
18:     {
19:         showStatus("J단추의 찰칵에 응답");
20:     }
21:}
```

프로그람설명

실례 8-26에서 13행은 JButton의 setToolTipText( )메소드를 호출하여 이 Swing 단추의 입력재촉정보를 하나의 문자렬로 설정한다. 마우스가 이 Swing단추에 1~2초 동안 머무른 후에 이 입력재촉정보를 마우스의 오른쪽아래에 자동출현시킨다. 또한 마우스가 Swing단추에서 벗어날 때 이 입력재촉정보는 자동제거된다.

그림 8-29는 실례 8-26의 실행결과이다.

그림 8-29. 실례 8-26의 실행결과

제9장. 망프로그람작성

제1절. 저층망통신의 실현

- 소케트는 처리들사이의 통신련결을 위한 망의 대면이다.
- TCP소케트
  Socket클라스: 의뢰기측의 소케트대면
  Socket(String host, int port) throws IOException
  Socket(InetAddress address, int port) throws IOException
  ServerSocket클라스: 봉사기측의 소케트대면
  ServerSocket(int port, int backlog) throws IOException
  Socket accept( ) throws IOException
- UDP소케트
  데이터그램통신을 실현하는 클라스
  DatagramPacket, DatagramSocket

Java를 리용하여 콤퓨터망의 저층통신을 실현한다는것은 Java프로그람을 리용하여 망통신규약이 규정한 기능과 조작을 실현한다는것이다. 이것은 Java망프로그람작성기술의 한 부분이다. 망통신규약의 종류는 많으므로 여기에서는 2가지 구체적인 규약의 Java프로그람작성만을 고찰한다.

9.1.1. 련결에 기초한 흐름식소케트

소케트(Socket)는 TCP/IP규약의 프로그람작성대면이다. 즉 Socket가 제공하는 API를 리용하면 TCP/IP규약을 작성하여 망통신을 실현할수 있다.

1) InetAddress클라스

우선 Java망프로그람작성에서 자주 사용하는 클라스를 소개한다. InetAddress클라스는 주로 콤퓨터망에서의 서로 다른 가입자들을 구별하는데 쓰인다. 매개 콤퓨터는 자기의 주소를 가지고있다. 매 InetAddress객체는 IP주소, 주콤퓨터이름 등 정보를 가지고있다. 아래의 례제는 주콤퓨터를 리용하여 망에서의 콤퓨터에 대한 IP주소를 현시하고있다.

실례 9-1

```
Example 9-1 MyIPAddress.java
1: import java.net.*;    //InetAddress클라스가 있는 패키지를 인입
2: public class MyIPAddress
3: {
4:      public static void main(String args[])
5:      {
6:      try{
7:          if (args.length == 1)
8:          {   // InetAddress클라스의 정적메쏘드 호출, 주콤퓨터이름을 리용하여 객체창조
9:              InetAddress ipa = InetAddress.getByName(args[0]);
10:             System.out.println("Host name:" + ipa.getHostName()); // 주콤퓨터이름얻기
11:             System.out.println("Host IP Address:" + ipa.toString()); // IP주소얻기
12:             System.out.println("Local Host:" + InetAddress.getLocalHost());
13:         }
14:         else
15:             System.out.println("지령행파라메터로 주콤퓨터이름을 입력하십시오");
16:     }
17:     catch(UnknownHostException e) //InetAddress객체가 일으킬수 있는 례외를 창조
18:     {
19:         System.out.println(e.toString());
20:     }
21:   }
22:}
```

프로그람설명

실례 9-1의 프로그람은 getHostName( )메쏘드를 리용하여 지령행파라메터로 지정한 주콤퓨터의 IP주소를 찾고 현시한다. getLocalHost( )는 InetAddress클라스의 정적메쏘드로서 이 프로그람을 실행하는 콤퓨터의 이름을 얻는데 쓰인다. 만일 아래의 지령행파라메터

C:> java MyIPAddress sun.com

을 입력하면 프로그람의 실행결과는

Host name: sun.com
Host IP Address: sun.com/205.179.206.240
Local Host: sharp/192.168.0.41

이다.

InetAddress클라스를 사용하면 프로그람에서 IP주소대신에 주콤퓨터이름을 쓸수 있다. 그리하여 프로그람이 보다 능동적으로 읽기를 쉽게 할수 있다.

2) 흐름소케트의 통신기구

흐름소케트로 진행하는 통신은 련결에 기초한 통신이다. 즉 통신개시전에 우선 통신쌍방에 의해 신분을 확인하고 전용가상련결통로를 만들어 자료성분을 전송하며 통신결속시에는 원래 련결한 접속을 끊는다. 이 과정을 그림 9-1에서 보여주었다.

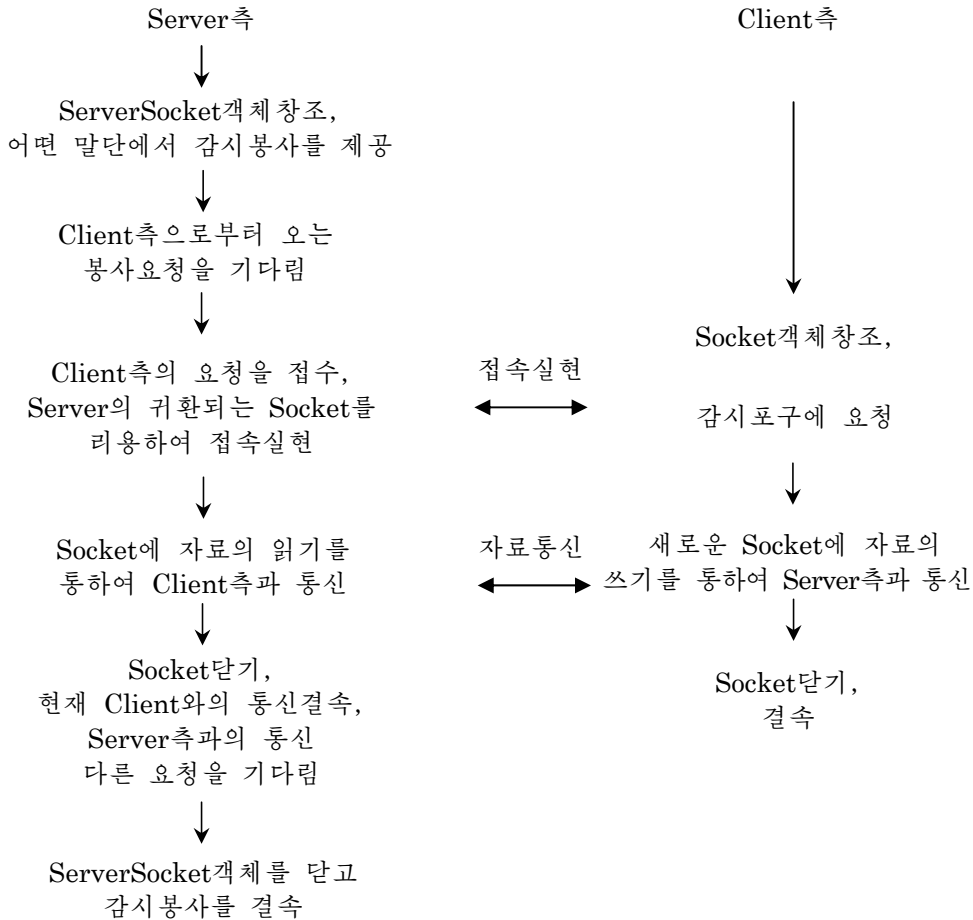



그림 9-1. 흐름식Socket통신과정

그림에서 Server측은 어떤 포구로 Client요구의 감시봉사를 제공하고 감시상태에 놓인다. Client측이 이 Server측에 봉사요구를 제공할 때 Server측과 Client측은 접속하여 자료를 보내는 통로를 만든다. 통신결속후 이 접속통로는 동시에 단절된다.

접속에 기초한 통신은 통신과정을 오유없이 정확히 담보할수 있다. 그러나 접속의 구성과 단절은 프로그람의 복잡성을 증가시키며 동시에 통신과정에서 시종일관 접속을 유지하는것 역시 체계의 기억기 등의 자원을 차지한다. 그러므로 집중적이고 련속적인

통신에만 적합하다.(례: 망에서의 담화 등) 불련속적이거나 실시간적인 요구가 강한 통신에 대하여서는 다음 절에서 소개하는 무접속데이터그램방식을 사용할수 있다.

3) Socket클라스와 ServerSocket클라스

그림 9-1에서 보여준 Socket클라스와 ServerSocket클라스는 Java에서 흐름식 Socket통신을 실현하는 중요한 클라스이다. ServerSocket객체를 창조하여 감시봉사를 진행하며 Socket객체를 창조하여 Cilent와 Server사이의 접속을 실현한다.

(1) ServerSocket클라스

아래의 명령문은 ServerSocket클라스를 창조하며 동시에 이 명령문을 실행하는 콤퓨터의 지정포구로 감시봉사를 진행한다.

```
ServerSocket MyListener = new ServerSocket(8000);
```

여기에서 감시봉사를 제공하는 포구번호는 8000이다. 한대의 콤퓨터는 동시에 여러개의 봉사를 제공할수 있으며 이것들은 봉사기들사이에 서로 다른 포구번호를 통하여 구별된다. 즉 서로 다른 포구번호는 서로 다른 봉사를 제공할수 있다. Client가 접속하려는 포구에 제공하려는 봉사를 접속시킬수 있다.

Client를 수시로 감시하기 위하여 아래의 명령문을 집행하여야 한다.

```
Socket LinkSocket = MyListener.accept( );
```

이 명령문은 ServerSocket객체의 accept( )메쏘드를 호출한다. 이 메쏘드는 Server측의 프로그람이 대기상태에 놓이게 하며 프로그람은 Cilent측으로부터 오는 요청을 기다리며 Client와 통신하는 Socket객체 LinkSocket를 귀환시킨다. 앞으로 Server프로그람은 이 Socket객체에 자료를 읽거나 쓰는것으로 원격의 Client에 자료의 읽기쓰기를 할수 있다.

감시를 결속하려는 때에는 아래의 명령문을 리용하여 ServerSocket객체를 닫으면 된다.

```
MyListener.close( );
```

(2) Socket클라스

Client측에서 Server측으로부터 정보와 기타 봉사를 얻으려고 할 때 Socket객체를 창조하여야 한다.

```
Socket MySocket= new Socket("ServerCoumputerName",8000);
```

Socket클라스의 구성자에는 두개의 파라메터가 있다. 첫번째 파라메터는 이미 접수한 Server콤퓨터의 주콤퓨터주소이며 두번째 파라메터는 이 Server에서 봉사를 제공하는 포구번호이다.

Socket객체의 창조가 실현된 후 Client와 Server사이에 접속을 실현할수 있으며 이 접속을 통하여 의뢰기와 봉사기사이에 자료가 전달될수 있다.

```
OutputStream SocketOs = MySocket.getOutputStream( );
InputStream SocketIs = MySocket.getInputStream( );
SocketOs.write(SocketIs.read( ));
```

우에서는 우선 Socket클라스의 getOutputStream( )과 getInputStream( )메쏘드를 리용하여 Socket에 자료를 읽고쓰는 입출력흐름을 얻는다. 마지막명령문은 Server측에 다시 되돌려보내는 기능을 수행한다. Server와 Client측의 통신결속이 끝나면 Socket클라스의 close( )메쏘드을 호출하여 Socket를 닫고 접속을 끊는다.

MySocket.close( );

4) 흐름식 Socket통신을 실현하는 Client측과 Server측프로그람의 작성과 실행

앞에서 소개한 내용을 종합하여 여기에서는 Socket통신을 실현하는 완전한 Java 프로그람을 작성한다.

실례 9-2-1

```
Example 9-2-1 MySocketServer.java
1: import java.io.*;
2: import java.net.*;
3: import java.awt.*;
4: import java.awt.event.*;
5: public class MySocketServer
6: {
7:      public static void main(String[] args)
8:      {
9:          ServerService MyServer = new ServerService(8000, 10);
10:     }
11:}
12:class ServiceThread extends Frame implements Runnable
13:{ //Client가 요청을 할 때 Server는 Frame을 창조
14:     ServerService FatherListener; //현재 통신토막처리를 창조하는 감시기객체
15:     Socket ConnectedClient; //현재토막처리에서 C/S통신을 담당하는 Socket객체
16:     Thread ConnectThread; //통신을 담당한 토막처리
17:     Panel ListenerPanel;
18:     TextArea ServerMeg; //정보현시창문의 본문구역
19:     public ServiceThread(ServerService sv, Socket s)
20:     {
21:         FatherListener = sv;
22:         ConnectedClient = s;
23:         ConnectThread = new Thread(this);
24:         setTitle("Answer Client"); //Server측 정보현시창문을 만들고 현시
25:         setLayout(new BorderLayout( ));
26:         ServerMeg = new TextArea(10, 50);
27:         add("Center", ServerMeg);
```

```
28:         setResizable(false);
29:         pack( );
30:         show( ); //봉사를 요구하는 Client측콤퓨터의 IP주소를 얻기
31:         InetAddress ClientAddress = ConnectedClient.getInetAddress( );
32:         ServerMeg.appendText("Client connected" + "from \n" +
                                            ClientAddress.toString( ) + ".\n");
33:     }
34:     public void run( ) //하위클라스는 Client와의 통신을 실현한다.
35:     {
36:         try{
37:             DataInputStream in = new DataInputStream(
                    new BufferedInputStream(ConnectedClient.getInputStream( )));
38:             PrintStream out = new PrintStream(
                 new BufferedOutputStream(ConnectedClient.getOutputStream( )));
39:             out.println("Hello! Wellcome connect to our server!\r");
40:             out.flush( ); //Client측에 정보를 출력
41:             String s = in.readLine( ); //Client측으로부터 정보를 읽기
42:             while(!s.equals("Bye"))
43:             {
44:                 ServerMeg.appendText("Client 측이 입력한 정보:\n" + s);
45:                 s = in.readLine( ); //Client측이 쓰기한 다음행 정보를 읽기
46:             }
47:             ConnectedClient.close( ); //Client측이 "Bye"를 쓰기하면 통신을 결속
48:         }
49:         catch(Exception e){ }
50:         FatherListener.addMeg("Client" + "closed." + "\n");
51:         dispose( ); //현재통신 Frame을 닫기
52:     }
53:}
54:class ServerService extends Frame //봉사기측의 감시기창문
55:{
56:     ServerSocket m_sListener;   //감시기
57:     TextArea ListenerMeg;       //정보를 현시하는 감시기창문
58:     public ServerService(int Port, int Count)
59:     {
60:         try{
61:             m_sListener = new ServerSocket(Port, Count); //감시봉사구축
62:             setTitle("Server Listener");
63:             this.addWindowListener(new WinAdpt( ));
```

```
64:             setLayout(new BorderLayout( ));
65:             ListenerMeg = new TextArea("감시봉사는 이미 기동\n", 10, 50);
66:             add("Center", ListenerMeg);
67:             setResizable(false);
68:             pack( );
69:             show( );
70:             while(true)
71:             {
72:                 Socket Connected = m_sListener.accept( );
                                                //Client측으로부터 온 요청을 접수
73:                 InetAddress ClientAddress = Connected.getInetAddress( );
74                 ListenerMeg.appendText("Client" + "connected" +
75:                                         "from \n" + ClientAddress.toString( ) +".\n");
76:
77:                 ServiceThread MyST = new ServiceThread(this, Connected);
78:                 MyST.ConnectThread.start( );     //새로운 토막처리기동
79:             }
80:         }
81:         catch(IOException e){ }
82:     }
83:     public void addMeg(String s)   //감시기창문에서 정보를 추가
84:     {
85:         ListenerMeg.appendText(s);
86:     }
87:}
88:class WinAdpt extends WindowAdapter
89:{
90:     public void windowClosing(WindowEvent e)
91:     {
92:         ((Frame)e.getWindow( )).dispose( );
93:         System.exit(0);
94:     }
95:}
```

실례 9-2-2

```
Example 9-2-2 MyClient.java
1: import java.awt.*;
2: import java.awt.event.*;
3: import java.net.*;
4: import java.io.*;
5: public class MyClient extends Frame implements ActionListener
6: {
7:      Socket ClientSocket;
8:      PrintStream os;
9:      DataInputStream is;
10:     String s;
11:     Label Mylabel = new Label("주콤퓨터가 제공하는 정보를 사용하는것을 환영합니다.");
12:     TextArea textArea;
13:     Button MyButton = new Button("송신");
14:     public MyClient( )
15:     {
16:         setTitle("Client Window");
17:         setLayout(new BorderLayout( ));
18:         this.addWindowListener(new WinAdptClient(this));
19:         MyButton.addActionListener(this);
20:         textArea = new TextArea(20, 50);
21:         add("North", Mylabel);
22:         add("South", MyButton);
23:         add("Center", textArea);
24:         setResizable(false);
25:         pack( );
26:         show( );
27:         connect( ); //Server측과 통신련결
28:     }
29:     public void connect( )
30:     {
31:         try{
32:             ClientSocket = new Socket("Ym", 8000); //Server콤퓨터의 8000포구에 련결
33:             os = new PrintStream(
                   new BufferedOutputStream(ClientSocket.getOutputStream( )));
34:             is = new DataInputStream(
                      new BufferedInputStream(ClientSocket.getInputStream( )));
```

```
35:            s = is.readLine( ); //Server측으로부터 자료읽기
36:            textArea.appendText(s + "\n");
37:        }
38:        catch(Exception e){ }
39:    }
40:    public void actionPerformed(ActionEvent e)
41:    {      //단추를 찰칵할 때 Server측에 정보를 송신
42:        if(e.getSource( ) == MyButton)
43:        {
44:            try{
45:                os.print(textArea.getText( ));
46:                os.flush( );
47:            }
48:            catch(Exception e1){ }
49:        }
50:    }
51:    public static void main(String args[])
52:    {
53:        new MyClient( );
54:    }
55:}
56:class WinAdptClient extends WindowAdapter
57:{
58:    MyClient m_Parent;
59:    WinAdptClient(MyClient p)
60:    {
61:        m_Parent = p;
62:    }
63:    public void windowClosing(WindowEvent e)
64:    {
65:        try{   //창문을 닫기전에 먼저 Server측에 결속정보를 송신
66:            m_Parent.os.println("Bye");
67:            m_Parent.os.flush( );
68:            m_Parent.is.close( );
69:            m_Parent.os.close( );
70:            m_Parent.ClientSocket.close( );
71:            m_Parent.dispose( );
72:            System.exit(0);
73:        }catch(Exception ex){ }
74:    }
75:}
```

프로그람설명

이 프로그람은 Server측에 의하여 실시간적으로 제공되는 정보봉사를 진행한다. 우선 Server측의 8000포구에서의 감시봉사를 구축한다. Client가 지정포구 8000에 접속할 때마다 Server측은 새로운 토막처리를 구축하여 이 Client와의 통신을 전문적으로 처리한다. 즉 Client측에 문자렬정보를 쓰기하고 Client측으로부터 정보를 읽어들여 Server에 현시한다. 또한 Client측 프로그람은 Server측에 접속한 다음에 먼저 Server측이 전달하는 정보를 접수하고 다음에 다시 Server측에 정보를 쓰기한다. 만일 Client가 Server측에 문자렬 Bye를 쓴 다음 통신을 결속해야 한다면 두 부분프로그람은 각각 자기의 Socket객체를 닫기하며 Server프로그람은 동시에 이 Client와의 통신토막처리를 닫는다.

9.1.2. 무접속데이터그램

흐름식Socket는 정확한 통신을 실현할수 있으나 자원을 많이 차지한다. 한편 실시간적인 영향을 받는(례: E−mail을 접수하거나 발송하는것 등)데서는 접속을 유지하는 흐름식통신이 적합하지 않으며 이때에는 데이터그램방식을 사용하여야 한다.

데이터그램은 접속하지 않는 원격통신봉사이다. 자료는 독립적인 파케트를 단위로 하여 발송되며 전송순서와 내용의 정확성은 보장하지 못한다. 데이터그램 Socket를 UDP소케트라고 한다. 그것은 접속을 구축하고 단절할 필요가 없이 정보를 직접 지정한 목적지에까지 보내며 흐름식Socket에 비해 사용하기가 어느정도 간단하다.

Java에서 무접속데이터그램통신에 쓰는 클라스에는 2가지가 있다. 즉 DatagramPacket클라스와 DatagramSocket클라스이다. 여기서 DatagramPacket클라스는 자료 등의 정보를 읽어들이며 DatagramSocket클라스는 데이터그램의 발송과 접수과정을 실현한다.

1) DatagramPacket클라스와 DatagramSocket클라스

(1) DatagramPacket클라스

DatagramPacket클라스의 구성자에는 2개가 있다.

```
public DatagramPacket(byte ibuf[], int ilength);
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport);
```

첫번째 함수는 데이터그램을 접수하는 객체를 창조하는데 쓰인다. 그것의 2개 파라메터는 각각 접수하는 자료부분의 바이트배렬과 바이트배렬의 길이를 의미한다. 두번째 함수는 원격체계에 보내는 데이터그램을 창조하는데 쓰인다. 첫번째 파라메터 ibuf는 이미 발송한 코드를 보관한 후의 데이터그램문의 바이트배렬이며 두번째 파라메터 ilength는 바이트배렬의 길이(데이터그램의 크기)를 지정한다. 세번째 파라메터는 발송하는 데이터그램의 목적지(접수자의 IP주소)를 지정한다. 마지막 파라메터 iport는 이 데이터그램을 목적주콤퓨터의 어느 포구에 발송하는가를 표시한다.

(2) DatagramSocket클라스

DatagramSocket클라스에는 3개의 구성자가 있다.

```
public DatagramSocket( );
public DatagramSocket(int port);
public DatagramSocket(int port, InetAddress localAddr);
```

첫번째 함수는 데이터그램Socket를 창조하는데 리용한다. 그것은 주콤퓨터의 임의의 리용가능한 포구에 접속한다. 두번째 함수는 지정한 포구에서 데이터그램Socket 객체를 창조한다. 세번째 함수는 지정한 IP주소를 가진 주콤퓨터에서 데이터그램 Socket를 창조하는데 쓰인다. 그의 두번째 파라메터는 어느 IP주소를 사용하였는가를 지정한다. 이 3개의 구성자는 IOException례외를 던지며 DatagramSocket클라스객체를 창조할 때 발생할수 있는 례외현상을 조종하는데 쓰인다.

receive( )와 send( )는 DatagramSocket클라스에서 데이터그램소케트가 데이터그램접수와 전송을 실현하는데 쓰이는 메쏘드이다.

```
public synchronized void receive(DatagramPacket p) throws IOException
public void send(DatagramPacket p) throws IOException
```

여기서 receive( )메쏘드는 프로그람의 토막처리가 줄곧 막기상태에 있게 하며 현재Socket로부터 데이터그램문, 발송자 등의 정보를 접수한다. 접수한 정보는 receive( )메쏘드의 파라메터 DatagramPacket객체 p에 저축한다. 주의하여야 할것은 데이터그램은 신용할수 없는 자료통신방식이므로 receive( )메쏘드가 DatagramPacket 객체 p의 데이터그램문을 지정한 IP주소의 주콤퓨터의 지정포구에 반드시 발송할수 있는것이 아니라는것이다. send( )메쏘드는 DatagramPacket객체 p에 포함된 데이터그램문을 지정한 IP주소의 주콤퓨터의 지정포구에 발송한다. 이 두 메쏘드는 입출력례외를 발생할수 있으므로 IOException례외를 던질수 있다.

2) UDP의 프로그람작성과 실현

(1) 데이터그램의 발송과정은 아래의 단계로 간단히 표현할수 있다.

① DatagramPacket객체를 창조한다. 여기에는 다음과 같은 정보가 포함된다.

- 발송하려는 자료
- 데이터그램배렬의 길이
- 목적주콤퓨터의 IP주소와 목적포구번호

② 지정 또는 리용가능한 주콤퓨터포구에서 DatagramSocket객체를 창조한다.

③ DatagramSocket의 send( )메쏘드를 호출하고 DatagramPacket객체를 파라메터로 하여 데이터그램을 발송한다.

(2) 데이터그램의 접수과정은 아래의 단계로 간단히 표현할수 있다.

① 데이터그램을 접수하는데 쓰이는 DatagramPacket객체를 창조한다. 여기서 빈자료완충기와 데이터그램배렬의 길이를 지정한다.

② 지정 또는 리용가능한 주콤퓨터포구에서 DatagramSocket객체를 창조한다.

③ DatagramSocket객체의 receive( )메쏘드를 호출하고 DatagramPacket객체를 파라메터로 하여 데이터그램을 접수한다. 접수한 정보에는 다음과 같은것이 포함된다.

· 접수한 데이터그램문의 내용
· 발송측의 주콤퓨터의 IP주소
· 발송측의 주콤퓨터의 발송포구번호

아래는 데이터그램Socket를 통하여 데이터그램을 발송접수하는 례이다. 실례에서는 봉사기를 우편봉사기로 설정하고 준비된 시각에 의뢰기로부터 작성한 우편을 접수한다. 우편을 받으면 그것을 발송한 의뢰기에 확인정보를 전송한다. 한편 의뢰기는 주콤퓨터이름과 포구를 이미 알고있는 봉사기에 우편을 보낸 후 봉사기가 접수하였다는 확인정보를 기다린다.

실례 9-3-1

```
Example 9-3-1 UDPServerService.java
1: import java.io.*;
2: import java.net.*;
3:
4: public class UDPServerService //봉사기토막처리의 주프로그람기동
5: {
6:     public static void main(String args[])
7:     {
8:         if(args.length < 1)
9:         {
10:            System.out.println("mail봉사에 쓰이는 포구번호를 입력하십시오");
11:            System.exit(0);
12:        }
13:        UDPServerThread MyUDPServer =
                           new UDPServerThread(Integer.parseInt(args[0]));
14:        MyUDPServer.start( );   //토막처리 기동
15:    }
16:}
17:class UDPServerThread extends Thread
18:{
19:    private DatagramSocket UDPServerSocket = null;
20:    public UDPServerThread(int Port)
21:    {
22:        try{ //봉사기측이 UDP를 송수신하는 DatagramSocket객체창조,
                                                   Port포구에서 UDP를 송수신
23:            UDPServerSocket = new DatagramSocket(Port);
```

```
24:             System.out.println("우편봉사감시기는 포구에 있습니다."
                                              + UDPServerSocket.getLocalPort( ) + "\n");
25:         }catch(Exception e){
26              System.err.println(e);
27:         }
28:     }
29:
30:     public void run( )  //토막처리의 주요조작
31:     {
32:         if(UDPServerSocket == null)
33:             return;
34:         while(true)
35:         {
36:             try{
37:                 byte dataBuf[] = new byte[512];
38:                 DatagramPacket ServerPacket;
39:                 InetAddress RemoteHost;
40:                 int RemotePort;
41:                 String Datagram, s;
42:                 ServerPacket = new DatagramPacket(dataBuf, 512);
43:                 UDPServerSocket.receive(ServerPacket);
44:                 RemoteHost = ServerPacket.getAddress( );
45:                 RemotePort = ServerPacket.getPort( );
46:                 Datagram = new String(ServerPacket.getData( ));
47:                 System.out.println("다음의 주콤퓨터가 보낸 우편을 접수 " +
                                    RemoteHost.getHostName( ) + ":\n"+Datagram);
48:                 Datagram = new String(RemoteHost.getHostName( ) +
                    ":\n MailServer " + InetAddress.getLocalHost( ).getHostName( )
                                                + "has alread get your mails.");
49:                 for(int i = 0; i < 512; i++)dataBuf[i] = 0;      //자료구역지우기
50:                     Datagram.getBytes(0, Datagram.length( ), dataBuf, 0);
51:                 ServerPacket = new DatagramPacket(dataBuf,
                                         dataBuf.length, RemoteHost, RemotePort);
52:                 UDPServerSocket.send(ServerPacket);
53:             }catch(Exception e){
54:                 System.err.println(e);
55:             }
56:         }
57:     }
```

```
58:     protected void finalize( )
59:     {    //프로그람결속시 끝내지 못한 소케트를 닫기
60:         if(UDPServerSocket != null)
61:         {
62:             UDPServerSocket.close( );
63:             UDPServerSocket = null;
64:             System.out.println("봉사기측의 데이터그램접속을 닫기");
65:         }
66:     }
67:}
```

실례 9-3-2

```
Example 9-3-2 ClientService.java
1: import java.io.*;
2: import java.net.*;
3: public class ClientService
4: {
5:      public static void main(String args[])
6:      {
7:          DatagramSocket UDPClientSocket; //UDP를 송수신하는데 쓰임
8:          DatagramPacket ClientPacket; //UDP의 내용을 보관하는데 쓰임
9:          InetAddress RemoteHost;
10:         int RemotePort;
11:         byte dataBuf[] = new byte[512];
12:         String Datagram, s;
13:         if(args.length < 3)
14:         {
15:             System.out.println("포구번호, 원격봉사기주콤퓨터이름,
                                    봉사기포구번호를 입력하십시오");
16:             System.exit(0);
17:         }
18:         try{
19:             UDPClientSocket = new DatagramSocket(Integer.parseInt(args[0]));
20:             RemoteHost = InetAddress.getByName(args[1]);
21:             RemotePort = Integer.parseInt(args[2]);
22:             Datagram = new String("This mail is from "
                          +InetAddress.getLocalHost( ).getHostName( )
```

```
                                +", give me a receipt\n if you can receive it, thank you!");
23:             Datagram.getBytes(0, Datagram.length( ), dataBuf, 0);
24:             ClientPacket = new DatagramPacket(dataBuf, 512, RemoteHost,
                                                                RemotePort);
25:             UDPClientSocket.send(ClientPacket); //원격봉사기에 정보를 발송
26:             for(int i = 0; i < 512; i++) dataBuf[i] = 0;
27:                 UDPClientSocket.receive(ClientPacket);
                                        //원격주콤퓨터의 귀환정보를 접수
28:             Datagram = new String(ClientPacket.getData( ));
29:             System.out.println("원격봉사기로부터 주콤퓨터 " + args[1]
                                + "는 다음의 응답정보를 접수하였습니다:");
30:             System.out.println(Datagram);
31:             UDPClientSocket.close( );
32:         }
33:         catch(Exception e){
34:             System.err.println(e);
35:         }
36:     }
37:}
```

제2절. 망자원에 대한 접근실현

- URL클라스는 URL주소로 원격자원에 접근하기 위한 클라스이다.
  URL(String url)
  URL(String url, String htmlfile)
  URL(String url, int port, String htmlfile)
- URLConnection은 원격거점에 정보를 전송하기 위한 클라스이다.
- Applet에서 망자원리용을 위한 메쏘드
  getAppletContext( ). showDocument(URL u)
  Image getImage(URL u)
  Image getImage(URL u, String s)
  getAudioClip( )

1절에서는 망에서 저층통신을 Java에서 어떻게 실현하는가를 고찰하였다. 이것은 체계가 제공하는 통신기능을 이미 만족시키고있으므로 보다 요구하는것은 Java를 리용하여 높은 층의 프로그람을 작성하는것이다. 즉 보통의 망응용프로그람을 작성하는것이다.

java.net패키지의 클라스 URL과 URLConnection은 망응용프로그람층의 HTTP 규약을 지원하므로 응용프로그람층의 망프로그람작성을 실현할수 있다.

9.2.1. URL클라스를 리용한 망자원의 접근

1) URL클라스

URL클라스의 객체는 URL주소를 표시하며 이 주소를 리용하여 원격자원에 접근할수 있다. URL주소는 일반적으로 《규약이름》, 《주콤퓨터이름》, 《경로파일이름》, 《포구번호》를 포함한다.

실례로 URL주소가

http://www.kcc.co.kp:80/index.html

이라면 규약이름은 http이며 주콤퓨터이름은 www.kcc.co.kp이고 경로파일이름은 index.html, 포구번호는 80이다.

규약이름과 포구번호사이에는 일정한 련관이 있다.

실례로 HTTP규약의 기정포구는 80이며 FTP규약의 기정포구는 21이다. 그러므로 URL주소에서 규약의 기정포구번호를 사용할 때에는 포구번호를 쓰지 않을수도 있다.

URL객체창조시 URL주소를 표시하여야 한다. 실례로 우의 URL주소는 아래와 같이 URL을 창조하는 명령문을 리용하여 표시할수 있다. 즉

URL MyURL1 = new URL("http://www.kcc.co.kp:80/");

URL MyURL2 = new URL("http", "www.kcc.co.kp", "index.html");

URL MyURL3 = new URL("http", "www.kcc.co.kp", 80, "index.html");

우의 3개의 구성자는 각각 서로 다른 방식으로 URL주소를 구성한다. 아래의 명령문은 URL클라스의 네번째 구성자를 사용한것이다.

URL MyURL4 = new URL(MyURL1, "support/kang.html");

이 구성자는 이미 있는 URL주소에 기초하여 상대경로파일을 제공한다. 이 구성자가 제공하는 URL주소는 다음과 같다.

http://www.kcc.co.kp:80/support/kang.html

2) URL클라스를 사용한 자원접근

URL클라스의 openStream( )메쏘드는 원격망정보를 얻는데 리용하는 전문적인 메쏘드이다. 이 메쏘드를 호출하면 입력흐름을 얻을수 있으며 이 입력흐름을 통하여 바이트단위로 원격거점의 정보를 읽어들일수 있다. 아래에 실례를 보여주었다.

실례 9-4

```
Example 9-4 getURLMeg.java
1: import java.net.*;
2: import java.io.*;
3: public class getURLMeg
4: {
5:      public static void main(String args[])
6:      {
7:          String s;
8:          try
9:          {
10:             URL MyURL = new URL("http://www.kcc.co.kp/"); //URL객체창조
11:             BufferedReader dis = new BufferedReader(
                        new InputStreamReader(MyURL.openStream( )));
12:             while((s = dis.readLine( )) != null)
                        //URL객체가 있는 곳으로부터 정보를 얻고 현시
13:             {
14:                 System.out.println(s);
15:             }
16:         }
17:         catch(MalformedURLException e) //URL객체가 일으킬수 있는 례외를 창조
18:         {   System.out.println("URL in wrong form, check it again.");}
19:         catch(IOException e)
20:         {   System.out.println("IO Exception occurred when get information."); }
21:     }
22:}
```

URL의 openStream( )메쏘드가 귀환시키는것은 InputStream클라스의 객체이므로 read( )메쏘드를 통하여 URL주소의 자원정보를 바이트단위로 차례로 읽어들인다.

조작을 간단히 하기 위하여 우의 프로그람에서는 BufferedReader를 사용하여 원시정보흐름에 대한 포장과 처리를 진행하고있으며 프로그람이 URL로부터 정보를 읽어들일수 있게 한다. 프로그람의 실행결과는 다음과 같다.

```
<html>
  <head>
    <meta http-equiv="Content-Type"content="text/html;charset=gb2312">
    <meta name="GENERATOR"content="Microsoft FrontPage 3.0">
    <title>Korean Computer Center</title>
  </head>
  …
</html>
```

실제상 URL이 지정하는 원격주콤퓨터의 HTML파일의 내용을 읽고있다. 여기에서 주의해야 할것은 URL객체를 창조할 때 주어진 주소정보가 정확하지 않으면 체계가 례외 MalformedURL을 던질수 있다는것이다. 그러므로 URL객체를 사용하는 프로그람은 이 례외를 처리하는데 주의하여야 한다.

9.2.2. URLConnection클라스의 사용

URL클라스를 사용하여 정보를 간단히 얻을수 있으나 만일 정보를 얻는 동시에 원격콤퓨터거점에 정보를 전송하려면 체계클라스서고의 클라스 URLConnection을 사용하여야 한다.

1) URLConnection클라스

URL주소정보를 리용하여 한개의 URL객체를 창조할 때 openConnection( )메쏘드를 호출하여 URL주소에 대응하는 URLConnection객체를 귀환시킬수 있다.

```
URL MyURL = new URL("http://www.kcc.co.kp/");
URLConnection MyURLConnection=MyURL.openConnection( );
```

URLConnection클라스는 망자원전달에 쓰이는 풍부한 메쏘드들을 포함하고있다. 실례로 getInputStream( )메쏘드는 URL거점으로부터 자료를 얻는 입력흐름을 귀환시키며 getOutputStream( )메쏘드는 URL거점에 자료를 보내는 출력흐름을 귀환시킨다. 여기서 입출력은 모두 HTTP규약에서 규정한 형식을 준수한다.

사실상 URLConnection객체를 구축하는 동시에 이 콤퓨터와 URL주소가 지정하는 원격거점에 이미 존재하는 HTTP규약의 접속통로를 구축한다. HTTP규약은 1회접속규약이며 발송하는 정보의 앞에 쌍방간의 신분을 확인하는 정보나 HTTP규약이 지정하는 부가정보를 추가하여야 한다. URLConnection객체를 구축한 다음에 자동적으로 접속되며 부가정보는 체계가 담당하므로 프로그람작성과정을 크게 간단화할수 있다.

2) URLConnection클라스의 사용

아래의 실례에서는 URLConnection클라스를 사용하여 원격주콤퓨터의 CGI응용프로그람을 호출한다.

실례 9-5

```
Example 10-5 TestURLConnection.java
1: import java.net.*;
2: import java.io.*;
3: public class TestURLConnection
4: {
5:      public static void main(String args[])
6:          {
7:          String s;
8:          try
9:          {
10:             URL MyURL = new URL("http://localhost/cgi/java/answer.class");
11:             URLConnection MyURLConnection = MyURL.openConnection( );
12:             PrintStream ps = new PrintStream(MyURLConnection.getOutputStream( ));
13:             BufferedReader dis = new BufferedReader(
                    new InputStreamReader(MyURLConnection.getInputStream( )));
14:             ps.println("Hello! This is a test.");
15:             ps.close( );
16:             while((s = dis.readLine( )) != null)
17:             {
18:                 System.out.println(s);
19:             }
20:             dis.close( );
21:         }
22:         catch(MalformedURLException e)
23:         { System.out.println("URL in wrong form, check it again."); }
24:         catch(IOException e)
25:         {   System.out.println("IO Exception occurred when get information."); }
26:     }
27:}
```

프로그람설명

실례 9-5에서 10, 11행을 집행하면 원격URL거점과의 HTTP접속이 완성된다. 이것은 열람기의 주소란에 http://localhost/cgi/java/answer.class를 입력하는것과 같다. 여기서는 HTTP규약의 CGI기능을 리용하여 원격주콤퓨터 localhost에 보존되여 있는 cgi등록부의 java프로그람 answer.class를 호출하여 실행한다. 여기의 cgi등록부는 가상등록부이다. 이에 대응하는 실제등록부가 E:\httpServer\javaCgi라면 우의 접속은 localhost에서 아래의 명령을 집행하는것과 같다. 즉

E:\http Server\javaCgi\java answer

여기서 프로그람 answer의 기능은 문자렬지령행파라메터를 접수하고 이 문자렬을 복사한 후에 출력하는것이며 이때 문자렬파라메터는 URLConnection의 출력흐름 ps를 통하여 원격주콤퓨터에 보내진다.

14행을 집행하면 열람기의 주소란에 아래의 정보를 건입력하는것과 같다.

"http://localhost/cgi/java/answer.class" + "Hello! This is a test"

원격주콤퓨터에서 CGI프로그람의 실행결과 문자렬 "Hello! This is a test"가 출력된다. 이 문자렬은 주콤퓨터의 프로그람에 의해 URLConnection의 입력흐름을 통하여 다시 읽어들여 현시한다. 프로그람실행결과는 아래와 같다.

```
C:>java TestURLConnection
Hello! This is a test.
```

만일 원격주콤퓨터로부터 어떤 격식을 가지고있는 정보를 얻고 이러한 정보를 자동해석하려면 URLConnection의 getConnect( )메쏘드를 사용할수 있다. 아래의 실례에서는 이 메쏘드를 사용하여 원격주콤퓨터에서 GIF도형파일을 얻고 Applet에 현시하였다.

실례 9-6

```
Example 9-6 TestURLImage.java
1: import java.applet.Applet;
2: import java.awt.*;
3: import java.net.*;
4: import java.io.*;
5: public class TestURLImage extends Applet
6: {
7:       public void paint(Graphics g)
8:       {
9:            Object o;
10:           try
11:           {
12:                URL MyURL = new URL("http://www.kcc.co.kp/background.gif");
```

```
13:            URLConnection MyURLConnection = MyURL.openConnection( );
14:            if((o = MyURLConnection.getContent( ))instanceof Image)
15:                g.drawImage((Image)o, 0, 0, this);
16:        }
17:        catch(MalformedURLException e)
18:        { System.out.println("URL in wrong form, check it again."); }
19:        catch(IOException e)
20:        { System.out.println("IO Exception occurred when get information.");}
21:    }
22:}
```

9.2.3. Applet의 메쏘드를 리용한 망자원의 접근

Applet클라스에서도 역시 망자원에 접근하는데 쓰이는 메쏘드를 정의하고있다. 지정페지의 접근을 포함하여 열람기에 현시하고 URL이 지정하는 화상과 음성을 얻으며 원격주콤퓨터의 음성을 얻은 후에는 직접 방영한다.

1) 지정페지의 접근

Applet의 getAppletContext( )메쏘드는 호출된 후 AppletContext클라스의 객체를 귀환시키며 객체의 필요한 메쏘드를 사용하여 열람기를 조종할수 있다. 실례로 AppletContext객체의 showDocument( )메쏘드를 호출하면 Applet를 실행하는 열람기를 조종할수 있으며 지정한 페지를 열람할수 있다.

실례 9-7

```
Example 9-7 appletBrowser.java
1: import java.applet.Applet;
2: import java.awt.*;
3: import java.net.*;
4: import java.awt.event.*;
5:
6: public class appletBrowser extends Applet
7: {
8:     public void init( )
9:     {
10:        this.addMouseListener(new MouseAdpt(this));
11:    }
12:    public void paint(Graphics g)
13:    {
```

```
14:        g.drawString("이 구역을 찰칵하면 열람기가 조선콤퓨터쎈터의
                                              홈페지에로 들어갑니다", 10, 20);
15:    }
16:}
17:class MouseAdpt extends MouseAdapter
18:{
19:    Applet m_Parent;
20:
21:    MouseAdpt(Applet p)
22:    {
23:        m_Parent = p;
24:    }
25:    public void mouseClicked(MouseEvent evt)
26:    {
27:        try
28:        {
29:            URL MyURL = new URL("http://www.kcc.co.kp/");
30:            m_Parent.getAppletContext( ).showDocument(MyURL);
31:        }
32:        catch(MalformedURLException e)
33:        { System.out.println("URL in wrong form, check it again."); }
34:    }
35:}
```

프로그람집행후 Applet구역을 찰칵하여 열람기가 망주소 http://www.kcc.co.kp 에로 이행하게 한다.

2) URL에 있는 화상을 얻기

Applet의 getImage( )메쏘드는 지정된 URL의 도형파일을 얻는데 리용한다. 이 메쏘드에는 2개의 재정의메쏘드가 있다.

```
Image getImage(URL u);
Image getImage(URL u, String s);
```

아래의 실례는 이 메쏘드를 사용하여 원격주콤퓨터로부터 화상파일을 얻으며 Applet에 현시한다.

실례 9-8

```
Example 9-8 getImage.java
1: import java.net.*;
2: import java.awt.*;
3: import java.applet.Applet;
4: public class getImage extends Applet
5: {
6:      Image MyImage;
7:      public void init( )
8:      {
9:          MyImage = getImage(getDocumentBase( ), "background.gif");
10:         repaint( );
11:     }
12:     public void paint(Graphics g)
13:     {
14:         g.drawImage(MyImage, 0, 0, this);
15:     }
16:}
```

실례에서는 getDocumentBase( )메쏘드를 리용하여 Applet의 HTML파일이 있는 URL주소의 background.gif화상파일을 현시한다.

3) URL에 있는 음성을 얻기

Applet의 getAudioClip( )메쏘드는 지정된 URL의 .au음성파일을 얻을수 있으며 또한 play( )메쏘드를 리용하여 망에서 음성파일을 직접 방영할수 있다. 아래에 실례를 보여주었다.

실례 9-9

```
Example 9-9 playSound.java
1: import java.net.*;
2: import java.awt.*;
3: import java.awt.event.*;
4: import java.applet.*;
5: public class playSound extends Applet
6: {
7:      AudioClip Myau;   //AudioClip는 java.applet.*의 대면이다
```

```
8:      public void init( )
9:      {
10:         Myau = getAudioClip(getDocumentBase( ), "cuckoo.au");
11:         this.addMouseListener(new MouseAdpt(this));
12:     }
13:     public void paint(Graphics g)
14:     {
15:         g.drawString("마우스가 Applet에 들어가면 첫번째 음성파일을 연주하고
            \n" + "Applet에서 탈퇴하면 두번째 파일을 연주합니다.", 10, 100);
16:     }
17:}
18:class MouseAdpt extends MouseAdapter
19:{
20:     Applet m_Parent;
21:
22:     MouseAdpt(Applet p)
23:     {
24:         m_Parent = p;
25:     }
26:     public void mouseEntered(MouseEvent e)
27:     {
28:         ((playSound)m_Parent).Myau.play( );
29:     }
30:     public void mouseExited(MouseEvent e)
31:     {
32:         m_Parent.play(m_Parent.getDocumentBase( ), "drwclose.au");
33:     }
34:}
```

프로그람설명

이 프로그람에서는 2개의 메쏘드를 사용하여 음성을 방영하였다. 하나는 AudioClip객체의 play( )메쏘드이며 다른 하나는 Applet객체의 play( )메쏘드이다. 사용자가 마우스를 Applet구역에로 이동시킬 때 첫번째 메쏘드를 사용하여 첫번째 음성파일을 방영하며 마우스가 Applet구역을 벗어나면 두번째 음성파일을 방송한다. 주의해야 할것은 Java프로그람이 방송할수 있는 음성파일은 모두 .au라는 확장자가 붙은 음성오림파일이라는것이다.

찾아보기